NPS52-82-011

NAVAL POSTGRADUATE SCHOOL

Monterey, California

TOP-DOWN SYNTHESIS OF
SIMPLE DIVIDE AND CONQUER ALGORITHMS

Douglas R. Smith

November 1982

Prepared for:
Chief of Naval Research
Arlington, Va 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-82-011	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Top-Down Synthesis of Simple Divide and Conquer Algorithms	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Douglas R. Smith	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01-10, N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	12. REPORT DATE: November 1982
	13. NUMBER OF PAGES: 103
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Chief of Naval Research, Arlington, Va 22217	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

automatic programming, program synthesis, deduction, divide and conquer, top-down programming

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A new method is presented for the deductive synthesis of computer programs. The method takes as given a formal specification of a user's problem. The specification is allowed to be incomplete in that some or all of the input conditions may be omitted. A completed specification plus a computer program are produced by the method. Synthesis involves the top-down decomposition of the user's problem into a hierarchy of subproblems. Solving each of these subproblems results in the synthesis of a hierarchically structured program. The program is guaranteed to satisfy the completed specification and to terminate on

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

all legal inputs.

In this paper we present a framework for a top-down synthesis process, explore the structure of a class of divide and conquer algorithms, and present a method for the top-down synthesis of algorithms in this class. Detailed derivations of four sorting algorithms are presented.

Top-Down Synthesis of Simple Divide and Conquer Algorithms[1]

Douglas R. Smith
Naval Postgraduate School
Monterey, California
12 November 1982

ABSTRACT

A new method is presented for the deductive synthesis of computer programs. The method takes as given a formal specification of a user's problem. The specification is allowed to be incomplete in that some or all of the input conditions may be omitted. A completed specification plus a computer program are produced by the method. Synthesis involves the top-down decomposition of the user's problem into a hierarchy of subproblems. Solving each of these subproblems results in the synthesis of a hierarchically structured program. The program is guaranteed to satisfy the completed specification and to terminate on all legal inputs.

In this paper we present a framework for a top-down synthesis process, explore the structure of a class of divide and conquer algorithms, and present a method for the top-down synthesis of algorithms in this class. Detailed derivations of four sorting algorithms are presented.

---

[1] The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Table of Contents

1. Introduction

Program synthesis is the task of automatically constructing a computer program from a description of the problem it is intended to solve. Although the prospect of a program synthesis system replacing human programmers is a long way off there are several near-term benefits to research on such systems. First, a program synthesis system requires considerable amounts of knowledge about programming and about the reasoning processes involved in synthesis. Before constructing such a system we are forced to formalize our knowledge, seeking a degree of explicitness not normally required by human programmers. This process can lead to deep insights into aspects of programming which were only loosely organized and intuitively understood previously. Properly formulated, such knowledge can contribute to a science of programming useful both to human programmers and automated programming systems. Second, as aspects of the programming task become better understood, it is natural to mechanize them, making it easier for human programmers to do their job. Thus, we expect to see special-purpose program synthesizers and programming aids become available long before the advent of general-purpose synthesizers.

In this paper we outline a program synthesis system and provide in detail some of the knowledge about programming required by such a system. Our basic approach to program synthesis is a form of top-down design - a technique well-known in software engineering circles but not previously formalized to the extent that it could be automated. Top-down design works as follows: given a description of a problem we decompose it into descriptions of subproblems in such a way that solutions for the subproblems can be assembled into a solution for the original problem. We then apply top-down design to each of the subproblem descriptions. The decomposition process terminates in primitive problem descriptions which are solved directly, without decomposition into subproblems. A solution to the original problem is then formed by composing solutions to subproblems according to the structure of the subproblem hierarchy.

One of the principal difficulties in top-down design is knowing how to decompose a problem into subproblems. At present general knowledge of this kind (see for example [18]) is intuitive and not in a form suitable for automation. Rather than attempt to formalize this general knowledge we focus on special ways to decompose a problem. In particular we present a theory concerning the structure of divide and conquer algorithms and show how to decompose a problem with respect to that structure.

The principle underlying divide and conquer algorithms can be simply stated: if the problem posed by a given input is sufficiently simple we solve it directly, otherwise we decompose it into subproblems, solve the subproblems, then compose the resulting solutions. The process of decomposing the input problem and solving the subproblems gives rise to the term "divide and conquer" although "decompose, solve and compose" would be more accurate. One form of divide and conquer is expressed in an ad-hoc language in Figure 1. We actually employ a more general schema in this paper.

The reader will notice that the divide and conquer principle and top-down design are similar in nature. Both rely on a collection of operators for decomposing problems into subproblems. These operators may work on some problems but not on others. In top-down design processes we often lack knowledge about which, if any, of the alternative operators will work on a given problem, so we must try each operator in turn; that is, we search. Divide and conquer algorithms, in contrast, have an inexpensive test to determine which operator applies to a given problem, thus they do not rely on search. We use the term simple divide and conquer for algorithms which have only a single decomposition operator.

We chose to explore the synthesis of divide and conquer algorithms for several reasons:

1. Structural Simplicity - Divide and conquer is perhaps the simplest program

---

```
Divide_and_Conquer(x) = if Primitive(x)
                        then Divide_and_Conquer := Direct_Solution (x)
                        else begin
                           (x1,x2) := Decompose(x);
                           y1 := Divide_and_Conquer(x1);
                           y2 := Divide_and_Conquer(x2);
                           Divide_and_Conquer := Compose(y1,y2)
                           end
```

Figure 1. A divide and conquer program schema

---

structuring technique which does not appear as an explicit control structure in current programming languages. Our description of the structure of divide and conquer algorithms is based on a view of them as computational homomorphisms between algebras on their input and output domains. Careful choice of programming language constructs allows us to express divide and conquer algorithms concisely and in accord with their essential structure as a computational homomorphism.

2. Computational Efficiency - Often algorithms of asymptotically optimal complexity arise from the application of the divide and conquer principle to a problem. Fast approximate algorithms for NP-hard problems frequently are based on the divide and conquer principle.

3. Ubiquity in Programming Practice - Divide and conquer algorithms are common in programming, especially when processing structured data objects such as arrays, lists, and trees. Current textbooks on the design of algorithms standardly present divide and conquer as a fundamental programming technique [1].

The basic concepts underlying our approach to program synthesis are presented in Section 2. While we have attempted to make this paper self-contained, some knowledge of first-order logic and automatic theorem proving techniques is presumed in Section 2.5. A system for top-down program synthesis is outlined in Section 3 together with the special knowledge needed to synthesize simple divide and conquer algorithms. A detailed illustration of the synthesis process is provided in Section 3.4 with the derivation of a selection sort. Less detailed but complete derivations are given in Section 4 of three other sorting algorithms. We distribute discussion of related research efforts and historical context among the appropriate sections.

2. Basic Concepts

2.1 Specifications

A program synthesis system requires as input a description of a problem to be solved. Specifications are a precise notation for describing the problem we desire to solve without necessarily indicating how to solve it. For example, the problem of sorting a list of natural numbers is may be specified as follows[2]

$$\text{Sort:}x = z \text{ such that } \text{Bag:}x = \text{Bag:}z \wedge \text{Ordered:}z$$
$$\text{where Sort: LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

Here the problem is named Sort which is a function from lists of natural numbers (denoted LIST(IN)) to lists of natural numbers. Naming the input x and the output z, the formula Bag:x = Bag:z $\wedge$ Ordered:z, called the output condition, expresses the conditions under which z is an acceptable output with respect to input x. Here Bag:x = Bag:y asserts that the multiset (bag) of elements in the list y is the same as he multiset of elements in x. Ordered:y is a predicate which holds exactly when the elements of list y are in nondecreasing order. Generally, a problem specification (or simply a specification) $\overline{\prod}$ has the form

$$\overline{\prod}:x = z \text{ such that } I:x \Rightarrow O:\langle x,z\rangle$$
$$\text{where } \overline{\prod}: D \rightarrow R.$$

We ambiguously use the symbol $\overline{\prod}$ to denote both the problem and its specification. Here the input and output domains are D and R respectively. The input condition expresses any properties we can expect of inputs to the desired program. Inputs satisfying the input condition will be called legal inputs. If an input does not satisfy the input condition then we don't care what output the program produces. The output condition O expresses the properties that an output object should satisfy. Any output object z such that $O:\langle x,z\rangle$ holds will be called a feasible output with respect to input x. More formally, a problem specification (specification) $\overline{\prod}$ is a 4-tuple $\langle D,R,I,O\rangle$ where

D is a set called the input domain,
R is a set called the output domain,
I is a relation on D called the input condition, and
O is a relation on $D \times R$ called the output condition.

---

[2] We use the notation f:x to denote the result of applying the function or program f to argument x.

The intention is that a program F satisfies a problem specification $\overline{\Pi}$ if on each legal input F computes feasible output. More formally, program F satisfies problem specification $\overline{\Pi} = \langle D,R,I,O \rangle$ if

$$\forall x \in D[I:x \Rightarrow O:\langle x,F:x\rangle] \qquad (2.1.1)$$

is valid in a suitable first-order theory.[3] We say $\overline{\Pi}$ is a complete specification of F if formula (2.1.1) is valid, otherwise it is an incomplete specification. Specification $\overline{\Pi} = \langle D,R,I,O \rangle$ is unsatisfiable if

$$\forall x \in D \; \forall z \in R \; [I:x \Rightarrow \sim O:\langle x,z\rangle]$$

is valid. In words, $\overline{\Pi}$ is unsatifiable if for each legal input there is no feasible output.

The definition of "satisfies" can be weakened slightly with the following ideas in mind. For several reasons we may not know what the input condition for a problem should be. Most importantly, the input conditions under which the output condition can be satisfied may be difficult to know ahead of time. That is, the class of inputs for which there exists feasible outputs may not be known or easily describeable. Also, within the computational or competence limits of a synthesis system it may not be possible to find a program which works on all legal inputs. In both cases we would like the synthesis system to "do the best it can" and yield a program F together with an input condition under which F is guaranteed to terminate with a feasible output. These considerations lead to the following definition: Program F satisfies specification $\overline{\Pi} = \langle D,R,I,O \rangle$ with derived input condition I' if

$$\forall x \in D \; [I':x \wedge I:x \Rightarrow O:\langle x,F:x\rangle]$$

is valid.

Note that a synthesis system employing this weaker concept of satisfaction can always generate a correct output; if the given problem is too hard it can always return a do-nothing program with the boolean constant FALSE as derived input condition. However, as we shall see, this concept of a derived input condition plays a more serious and integral role in our method in that they are used to construct certain predicates. Of course the synthesis of a program involves trying to make the derived input condition as weak as possible. We

---

[3] - A suitable first-order theory is discussed in Section 2.5. It is assumed in this paper that all predicates involved in a specification are total.

also note that a system based on this definition of satisfaction allows the user to ignore input conditions when formulating a specification. However, deriving an input condition may require considerable computation which could be saved if the user supplies a correct or nearly correct input condition initially.

As an example, suppose a program synthesis system is given the specification

$$\text{Select:}x = \langle a,z\rangle \text{ such that } a \leq \text{Bag:}z \wedge \text{Bag:}x = \text{Add:}\langle a,\text{Bag:}z\rangle$$
$$\text{where Select: LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N}).$$

Here we wish to split a list x into two components, a number a and a list z such that a is no larger than any element of z and the collection of elements in x is the same as the collection of elements in z with a added. This specification is incomplete in that there is no feasible output with respect to legal input nil (nil denotes the empty list). Our synthesis method would return a satisfying program (see Section 3.5.2) with derived input condition $x \neq \text{nil}$. This condition is then used as a guard on the invocation of Select.

2.2 Target Programming Language

We will express algorithms in a typed functional programming language FPL based on Backus' FP systems [2]. In a functional programming language programs are viewed as a hierarchy of functions. Such languages come equipped with a set of primitive functions and a set of combining forms which are used to create complex functions from simpler ones. FPL differs from the FP-system in Backus' paper by allowing data types, new combining forms called function products and nondeterministic conditionals, and a little syntactic sugar.

An FP-like system such as FPL can be described in terms of the following components:

1. a set of data types and a set of data objects
2. a set of primitive functions
3. an operation called application
4. a set of combining forms
5. a function definition mechanism.

Data Types and Data Objects

The data types of interest in this paper are $\mathbb{N}$ (natural numbers), LIST($\mathbb{N}$) (linear lists of natural numbers), and $\mathbb{B}$ (boolean values TRUE and FALSE). The symbol $\perp$, called bottom or undefined, is a data object and any elements of the preceeding data types are data objects. If $x_1,\ldots,x_n$ for $n \geq 0$ are data objects then the n-tuple $\langle x_1,\ldots,x_n\rangle$ is also a data object. If some object in a n-tuple is $\perp$ then the n-tuple is $\perp$; i.e., $\langle\ldots,\perp,\ldots\rangle = \perp$. For the purposes of specifying, discussing, and reasoning about programs we extend the usual equality relation so that it is defined when one of its arguments is $\perp$. The function Defined:x will be used to distinguish $\perp$ from other objects. For example, Defined:$\perp$ = FALSE, Defined:<1,3,5> = TRUE.

Application

The application of a function f to an object x, written f:x, denotes the object which results from applying f to x.

Functions

All functions map a data object to a data object. If a function requires n arguments for some $n \geq 0$, then it is applied to an n-tuple of objects. Similarly, if a function generates m outputs it returns an m-tuple of objects. Functions in the system are either primitive (supplied with the system) or functional forms (created from other functions by means of combining forms). The primitive functions of FPL are listed below in Figure 2 according to data type. For the natural numbers we have the usual addition function, denoted +, the comparison functions $<, \leq, =, \neq, \geq, >$, and the identity function, denoted Id. On the data type LIST($\mathbb{N}$) we use the functions First, which returns the first element in a list, Rest, which returns its input list minus the first element, Cons, which adds a number to the front of a list, Append, which concatenates two lists, Length, which returns the length of a list, and the identity function, denoted Id. Not included in the table are the selector functions $\underset{\sim}{1}$, $\underset{\sim}{2}$, etc. defined on tuples. For example, $\underset{\sim}{2}$:<3,(1,2,3),5> = (1,2,3), $\underset{\sim}{2}$:<1> = $\perp$. All functions in FPL are $\perp$-preserving in that f:$\perp$ = $\perp$ holds for each function in the system.

Combining Forms

Combining forms are used to create a new function from other functions. The following four combining forms will be used:

1. $\mathbb{B}$ (Boolean) example values: TRUE,FALSE

2. $\mathbb{N}$ (natural numbers) example values: 0, 1, 2,...

	name	examples
functions:	+	+:<3,5> = 8
	<, $\leq$, =, $\geq$, >, $\neq$	$\leq$:<3,5> = TRUE
		$\neq$:<3,5> = false
	Id (identity)	Id:3 = 3

3. LIST($\mathbb{N}$) (Lists of Natural Numbers) example values: nil = (), (1,2), (4)

	name	example
functions:	First	First:(2,5,3) = 2
		First:() = $\perp$
	Rest	Rest:(2,5,3) = (5,3)
		Rest:nil = $\perp$
	Cons	Cons:<2,(5,3)> = (2,5,3)
		Cons:<2,nil> = (2)
	Append	Append:<(2,4),(5,3,2)> = (2,4,5,3,2)
	Length	Length:(2,4,5,3) = 4
		Length:nil = 0
	Id	Id:(2,4,5,3) = (2,4,5,3)

Figure 2. Data Types and Primitive Functions in FPL

1. Composition - The composition of functions f and g is written f·g and is computed by first applying g to the data object then applying f to the result, so for all data objects x (f·g):x = f:(g:x).

For example: Length·Rest:(1,3,5) = Length:(Rest:(1,3,5))
= Length:(3,5)
= 2

2. Construction - If $f_1,\ldots,f_n$ are functions and x a data object then the construction of these functions is written $[f_1,\ldots, f_n]$ and is defined by

$$[f_1,\ldots,f_n]:x = \langle f_1:x,\ldots,f_n:x\rangle.$$

For example: [First,Rest]:(1,3,5) = <1,(3,5)>

3. Product - The product of unary functions $f_1,\ldots,f_n$, written $f_1 \times \ldots \times f_n$, is defined by

$$f_1 \times \ldots \times f_n:\langle x_1,\ldots,x_n\rangle = \langle f_1:x_1,\ldots,f_n:x_n\rangle.$$

for all data objects $x_1,\ldots,x_n$.

For example: Id $\times$ Length:<3,(1,3,5,7)> = <3,4>.

4. Nondeterministic Conditional - If $q_1,\ldots,q_n$ are boolean functions or constants and $f_1,\ldots,f_n$ are functions or data objects then

$$\text{if } q_1 \rightarrow f_1 \;[]\; \ldots \;[]\; q_n \rightarrow f_n \text{ fi}$$

is a nondeterministic conditional form [8]. During application to object x each of the boolean functions, called quards, are applied to x. If any of the quards evaluates to $\perp$, or if none of the quards evaluate to TRUE, then the form evaluates to $\perp$. Otherwise one of the guards, say q_i, which evaluates to TRUE is nondeterministically selected and the form evaluates to $f_i:x$.

For example,

$$\text{if } \leq \rightarrow \underset{\sim}{1} \;[]\; \geq \rightarrow \underset{\sim}{2} \text{ fi}$$

is a simple if-fi form mapping $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$ and computing the minimum of two natural numbers. On application to <2,3> the guard $\leq$:<2,3> evaluates to TRUE thus the form evaluates to $\underset{\sim}{1}$:<2,3> = 2. Note that on application to <3,3> both guards evaluate to TRUE thus either branch of the conditional can be taken. Although either branch can be taken the result is the same for this function.

Definitions

A definition is written l $\equiv$ r where l is the name of the function being defined and r is a functional form. For example, we can name the minimum function defined above:

$$\text{min} \equiv \text{if } \leq \rightarrow \underset{\sim}{1} \;[]\; \rightarrow \underset{\sim}{2} \text{ fi}$$

Hereafter we will sugar the above notation for the sake of readability by 1) allowing the left and right hand side of definitions to name their operands,

2) allowing binary boolean functions which are conventionally written in infix notation to be so expressed, 3) write

$$
\begin{array}{l}
\text{if} \\
\quad q_1{:}x \rightarrow f_1{:}x \ [] \\
\quad \cdots \\
\quad q_n{:}x \rightarrow f_n{:}x \\
\text{fi}
\end{array}
$$

for conditional forms, and 4) whenever possible replace selector functions on n-tuples by the name of the object or functional form which results from their application. Thus we will write the definition of the minimum function in the form:

$$
\begin{array}{l}
\text{Min}{:}\langle x,y\rangle \equiv \text{if} \\
\quad\quad x \leq y \rightarrow x \ [] \\
\quad\quad x \geq y \rightarrow y \\
\quad \text{fi}
\end{array}
$$

As a more complex example of a function in FPL consider the following recursive definition of Select, which was specified in the previous section. This function is a crucial component of a selection sort function and will be synthesized later. Given a nonempty list of natural numbers the job of Select is to split it into a 2-tuple containing the least element and the rest of the list.

$$
\begin{array}{l}
\text{Select}{:}x \equiv \text{if} \\
\quad\quad \text{Rest}{:}x = \text{nil} \rightarrow [\text{First, Rest}]{:}x \ [] \\
\quad\quad \text{Rest}{:}x \neq \text{nil} \rightarrow \text{Compose} \cdot (\text{Id} \times \text{Select}) \cdot [\text{First,Rest}]{:}x \\
\quad \text{fi}
\end{array}
$$

$$
\begin{array}{l}
\text{Compose}{:}\langle v_1,\langle v_2,z\rangle\rangle \equiv \text{if} \\
\quad\quad v_1 \leq v_2 \rightarrow \langle v_1,\text{Cons}{:}\langle v_2,z\rangle\rangle \ [] \\
\quad\quad v_1 \geq v_2 \rightarrow \langle v_2,\text{Cons}{:}\langle v_1,z\rangle\rangle \\
\quad \text{fi}
\end{array}
$$

Here is a simple evaluation of Select on the list (2,5,1,4)

```
Select:(2,5,1,4) = Compose·(Id X Select)·[First,Rest]:(2,5,1,4)
                 = Compose·(Id X Select):<2,(5,1,4)>
                 = Compose:<2,<1,(5,4)>>
                 = <1,Cons:<2,(5,4)>>
                 = <1,(2,5,4)>
```

where Select:(5,1,4) evaluates to <1,(5,4)> in a similar manner.

Select exemplifies the structure of simple divide and conquer algorithms. when Rest:x = nil then the problem is solved directly, otherwise the input is decomposed via the construction [First,Rest], recursively solved via the product (Id X Select), and the results composed via Compose.

There are several reasons for introducing FPL rather than using a known language such as LISP. First, programs in this functional language have a hierarchic structure which facilitates top-down program synthesis. To construct a function from a given specification we must know how to select an appropriate combining form and adapt it to the given problem. The adaptation involves finding functions for each of the slots in the combining form - either by supplying a primitive function or by deriving a specification for it. For the adaptation to be successful we must show that if the specifications for the component functions are satisfied then the resulting functional form will satisfy the original specification. One point to note here is that much of the knowledge needed by the synthesis process is related to the individual combining forms (i.e., how to adapt a certain combining form to a given specification) and the primitive functions (how to know when a primitive function satisfies a given specification). So the structure of the language provides a natural organizing framework for the programming knowledge required by a top-down program synthesizer.

In Section 3.2 we present a program schema for a class of divide and conquer algorithms and the programming knowledge needed to adapt it to a specification. It can be viewed as a derived combining form since it is expressed in terms of the primitive combining forms supplied with the language.

A second reason for introducing this language is that it allows an elegant formulation of divide and conquer programs. Compare, for example, the divide

and conquer program schema in Figure 1 with its expression in FPL:

```
DC:x ≡ if
          Primitive:x  →  Direct_Solution:x  []
         ~Primitive:x  →  Compose·(DC ⨯ DC)·Decompose:x
       fi
```

The language frees us from the need to overdetermine the order of evaluation of certain operations which are naturally independent. For example, in conditionals we are not forced to determine the order in which the guards are to be evaluated - they are conceptually evaluated in parallel. Also, the construction and product forms allow us to express processes which might otherwise require the storing of data and an arbitrary ordering of function evaluations. This conceptual parallelism is useful in expressing the functions synthesized in this paper and simplifies the synthesis process.

2.3 Program Termination and Well-Founded Orderings

Ensuring that a constructed program will terminate on all legal inputs (or more generally determining the input conditions under which it will terminate) is crucial to the usefulness of a synthesis method. The usual method for showing the termination of a recursive program depends on the existence of a well-founded ordering on the input domain.

A structure $\langle W, \succ \rangle$ where W is a set and $\succ$ is a binary relation on W is a well-founded set and $\succ$ is a well-founded ordering on W if:

1) $\succ$ is irreflexive: $u \not\succ u$ for all $u \in W$
2) $\succ$ is assymetric: if $u \succ v$ then $v \not\succ u$ for all $u,v \in W$
3) $\succ$ is transitive: if $u \succ v$ and $v \succ w$ then $u \succ w$ for all $u,v,w \in W$
4) there is no infinite descending sequence $u_0 \succ u_1 \succ u_2 \succ \ldots$ in W.

For example, $\mathbb{N}$ (natural numbers) with the usual 'greater than' relation > forms a well-founded set denoted $\langle \mathbb{N}, > \rangle$. More generally $\mathbb{N}^k$ (k-tuples of natural numbers) has a well-founded ordering denoted $>_k$ where:

$\langle n_1, \ldots, n_k \rangle >_k \langle n'_1, \ldots, n'_k \rangle$ iff there is some constant m, $1 \leq m < k$, such that
$n_i = n'_i$ for each $i < m$ and $n_m > n'_m$

So for all $k \geq 1$ $\langle \mathbb{N}^k, >_k \rangle$ is a well-founded set.

A recursive program P with input domain D can be shown to terminate on all inputs in the following way. First, a well-founded ordering $\succ$ is constructed on D. Then, we show that for any $x \in D$ P applied to x only generates recursive applications (calls) to inputs x' for which $x \succ x'$. There can be no infinite sequence $x_0, x_1, x_2 \ldots$ such that applying P to x_i results in the application of P to x_{i+1} for $i \geq 0$ since the well-founded ordering does not allow $x_0 \succ x_1 \succ x_2 \succ \ldots$. The above steps for ensuring program termination are an integral part of the synthesis method described below.

A program synthesis system will have knowledge of some standard well-founded sets such as $\langle \mathbb{N}, > \rangle$ and $\langle \mathbb{N}^k, >_k \rangle$ but it need not anticipate ahead of time all domains which will require well-founded orderings. Consequently a method for constructing well-founded orderings is needed. The following theorem asserts that if we have a domain E and a known well-founded set $\langle W, \succ \rangle$ then any function from E to W can be used to define a well-founded ordering on E.

Proposition 1. Let E be a set, let $\langle W, \succ_W \rangle$ be a well-founded set, and let $h: E \rightarrow W$ be a function from E into W. The relation $\succ_E$ defined by:

$$u \succ_E u' \text{ iff } h(u) \succ_W h(u')$$

is a well-founded ordering on E.

Proof: 1) $\succ_E$ is irreflexive - for any u, $h:u \not\succ_W h:u$, but then by definition $u \not\succ_E u$.

2) $\succ_E$ is assymetric - if $u \succ_E u'$ then $h(u) \succ_W h(u')$ and $h(u') \not\succ_W h(u)$ (by assymetry of $\succ_W$) thus $u' \not\succ_W u$.

3) $\succ_E$ is transitive - if $u \succ_E u'$ and $u' \succ_E u''$ then $h(u) \succ_W h(u')$ and $h(u') \succ_W h(u'')$. $h(u) \succ_W h(u'')$ follows by transitivity of $\succ_W$, then $u \succ_E u''$ follows by definition of $\succ_E$.

4) $\langle E, \succ_E \rangle$ has no infinite decreasing sequence - if $u_0 \succ_E u_1 \succ_E u_2 \succ_E \ldots$ then $h(u_0) \succ_W h(u_1) \succ_W h(u_2) \succ \ldots$ contradicting the well-foundedness of $\langle W, \succ_W \rangle$. QED

Proposition 1 enables us to establish a well-founded ordering on LIST($\mathbb{N}$) (list of natural numbers) by simply finding a function from LIST($\mathbb{N}$) to $\mathbb{N}$. A suitable primitive function is Length, so we may define

$$x \succ y \text{ iff } \text{Length:}x > \text{Length:}y$$

for all $x,y \in \text{LIST}(\mathbb{N})$. By Proposition 1 we conclude that $\langle \text{LIST}(\mathbb{N}), \succ \rangle$ is a well-founded set. Similarly, Length $\times$ Length (which maps $\text{LISTT}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$ to $\mathbb{N} \times \mathbb{N}$) can be used to construct a well-founded ordering $\succ_2$ on $\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$ where

$$\langle x,y \rangle \succ_2 \langle x',y' \rangle \text{ iff } \text{Length} \times \text{Length}:\langle x,y \rangle >_2 \text{Length} \times \text{Length}:\langle x',y' \rangle$$

$$\text{iff } \text{Length:}x > \text{Length}(x') \text{ or}$$

$$(\text{Length:}x = \text{Length}(x') \text{ and } \text{Length}(y) > \text{Length}(y'))$$

and so on.

Proposition 2: If $\langle W, \succ_W \rangle$ is a well-founded set then $\langle V_1 \times V_2 \times \ldots V_k, \succ_V \rangle$ where $V_i = W$ for some $i \in \{1,2,\ldots,k\}$ and $\succ_V$ is defined by

$$\langle u_1,u_2,\ldots u_k \rangle \succ_V \langle v_1,v_2,\ldots v_k \rangle \text{ iff } u_i \succ_W v_i$$

is also a well-founded set.

2.4 Many-Sorted Algebras

Algebraic concepts are playing an increasingly important role in formulating the fundamental notions of computer science. In this paper we show that divide and conquer algorithms can be usefully characterized in algebraic terms. In particular they can be viewed as homomorphisms between appropriately defined algebras on the input and output domains. Accordingly, the synthesis method described later involves the construction of these algebras. In this section we present the basic terminology of many-sorted algebras based on and extending the notation of [11,12].

For any $n \in \mathbb{N}$ let $\underline{n} = \{1,2,\ldots,n\}$. If A is a set, then A^+ will denote $A \cup \{\perp\}$ - the extension of A to include the symbol for undefinedness. As usual the cartesian product of sets $A_1, A_2, \ldots, A_n$ is written $A_1 \times A_2 \times \ldots \times A_n$ and denotes $\{\langle a_1,a_2,\ldots,a_n \rangle \mid a_i \in A_i \text{ for } i \in \underline{n}\}$. Parentheses are used for nesting so

$$A_1 \times (A_2 \times A_3) = \{\langle a_1,\langle a_2,a_3 \rangle \rangle \mid a_1 \in A_1,\ a_2 \in A_2,\ a_3 \in A_3\}$$

the set of 2-tuples whose first component belongs to A_1, and whose second component belongs to $A_2 \times A_3$.

Generally, we use the term simple many-sorted algebra to denote a collection of sets equipped with an operator defined on cartesian products of the sets. Let S denote a set of symbols called sorts. A simple S-sorted signature Σ of type $\langle w,\hat{s}\rangle$ where $w \in S^*$, $\hat{s} \in S$ is a set containing a single operator symbol of type $\langle w,\hat{s}\rangle$. Let $\langle A_s\rangle_{s \in S}$ be an S-indexed family of sets. If $w \in S^*$ and $w = w_1w_2\ldots w_n$ then A^w denotes the cartesian product $A_{w_1} \times A_{w_2} \times \ldots \times A_{w_n}$. A Σ-algebra A consists of a family of sets $\langle A_s\rangle_{s \in S}$ called the carriers of A, and an operator denoted σ_A where $\sigma_A:A^w \rightarrow A_{\hat{s}}$. $A_{\hat{s}}$ will be called the principal carrier of A. A Σ-algebra A will be written $A = \langle\{C_1,\ldots,C_k\},\{f\}\rangle$ where $\{C_1,\ldots,C_k\}$ are the carriers of A and f is its sole operator.

Let A and B be Σ-algebras and let $H = \langle h_s\rangle_{s \in S}$ be an S-indexed family of functions where for each $s \in S$, $h_s:A_s \rightarrow B_s$. If $w = w_1w_2\ldots w_n$ let h^w denote the product function $h_{w_1} \times h_{w_2} \times \ldots \times h_{w_n}$. Thus if $a \in A^w$ then

$$h^w:a = \langle h_{w_1}:a_1,\ h_{w_2}:a_2,\ \ldots,\ h_{w_n}:a_n\rangle.$$

$H = \langle h_s\rangle_{s \in S}$ is a ($\Sigma\Sigma$-)homomorphism from A to B if for each $a \in A^w$

$$h_{\hat{s}} \cdot \sigma_A:a = \sigma_B \cdot h^w:a. \qquad (2.4.1)$$

i.e. the diagram in Figure 4 commutes. A Σ-algebra will be called a composition algebra.

We illustrate this notation with examples relating to data structures and divide and conquer algorithms.

Example 2.4.1: Consider the sort set S = {a,b} and simple S-sorted signature Σ of

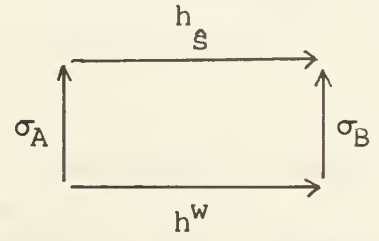


Figure 4: Commutative Diagram of a $\Sigma\Sigma$-homomorphism.

type <ab,b>. We describe two Σ-algebras called L and B as follows:

$$L = \langle \{\mathbb{N}, LIST(\mathbb{N})\}, \{Cons\} \rangle$$

$$B = \langle \{\mathbb{N}, BAGS(\mathbb{N})\}, \{Add\} \rangle$$

The carriers of L are $L_a = \mathbb{N}$ (natural numbers) and $L_b = LIST(\mathbb{N})$ (lists of natural numbers). The carriers of B are $B_a = \mathbb{N}$ (natural numbers) and $B_b = LIST(\mathbb{N})$ (lists of natural numbers). The operator symbol σ in Σ is interpreted in L as the function Cons, but in B is interpreted as Add (Add inserts a number into a bag of numbers); i.e. σ_L is Cons and σ_B is Add.

Example 2.4.2: There is a natural homorphism between algebras L and B defined in Example 2.4.1. Let h_a be Bag which maps a list of natural numbers x into the multiset of elements in x (e.g., Bag:(1,3,5,3,2) = {1,3,5,3,2}). Let h_b be the identity function Id. First, h_a and h_b have the correct domains and codomains:

$$Id: \mathbb{N} \rightarrow \mathbb{N} \quad (h_a: L_a \rightarrow B_a)$$

$$Bag: LIST(\mathbb{N}) \rightarrow BAGS(\mathbb{N}) \quad (h_b: L_b \rightarrow B_b).$$

Second, the homorphism condition (2.4.1) is satisfied: For each $a \in \mathbb{N}$ and $x \in LIST(\mathbb{N})$

$$Bag \cdot cons:\langle a,x \rangle = Add \cdot (Id \times Bag):\langle a,x \rangle$$

$$(\text{abstractly, } h_b \cdot \sigma_L:\langle a,x \rangle = \sigma_B \cdot (h_a \times h_b):\langle a,x \rangle).$$

The inverse Σ^{-1} of a simple S-sorted signature Σ of type $\langle w,\hat{s} \rangle$ is a set containing a single operator symbol of type $\langle \hat{s},w \rangle$. A Σ^{-1}-algebra A is a family of sets $\langle A_s \rangle_{s \in S}$ and an operator $\sigma_A: A_{\hat{s}} \rightarrow A^w$. Let A be a Σ^{-1}-algebra, B a Σ-algebra, and let $H = \langle h_s \rangle_{s \in S}$ be an S-indexed family of functions such that for each $s \in S$ $h_s:A_s \rightarrow B_s$. H is a $(\Sigma^{-1}\Sigma)$-homomorphism from A to B if for each $x \in A_{\hat{s}}$ such that $\sigma_A:x$ is defined

$$h_{\hat{s}}:x = \sigma_B \cdot h^w \cdot \sigma_A:x \qquad (2.4.2)$$

i.e., the diagram in Figure 5 commutes. A Σ^{-1}-algebra will be called a decomposition algebra.

Example 2.4.3: Consider a simple S-sorted signature Σ of type <ab,b>. Consider LS and LC which are Σ^{-1} and Σ-algebras respectively where:

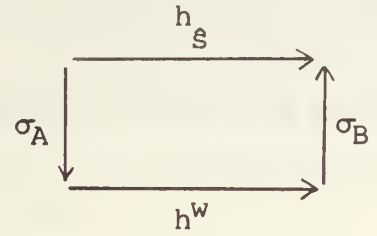


Figure 5: Commutative Diagram of a $\sum^{-1}\sum$-homomorphism.

$$LS = \langle \{ \mathbb{N}, LIST(\mathbb{N}) \}, \{Select\} \rangle$$

$$LC = \langle \{ \mathbb{N}, LIST(\mathbb{N}) \}, \{Cons\} \rangle$$

LS has carriers $LS_a = \mathbb{N}$ and $LS_b = LIST(\mathbb{N})$ and operator Select: $LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N})$. Select splits a list of natural numbers into its least element and the rest of the list as discussed earlier. LC has carriers $LC_a = \mathbb{N}$ and $LC_b = LIST(\mathbb{N})$ and operator Cons: $\mathbb{N} \times LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$. Letting h_b be the function Sort, which sorts a list of numbers, and h_a the identity function Id, we have a natural homomorphism from LS to LM. First, Sort and Id have the required domains and codomains:

$$Id: \mathbb{N} \rightarrow \mathbb{N} \qquad (h_a: LS_a \rightarrow LC_a)$$

$$Sort: LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}) \qquad (h_b: LS_b \rightarrow LC_b)$$

and the homomorphism condition (2.4.2) is satisfied: for any $x \in LIST(\mathbb{N})$ such that Select:x is defined.

$$Sort:x = Cons \cdot (Id \times Sort) \cdot Select:x.$$

Programmers will recognize this identity as the essence of a selection sort algorithm. It states that there are two ways to sort a list x: either apply Sort to x or decompose x via Select into a number a and list y, apply Sort to y yielding sorted list z then Cons a onto it.

It will be useful to relax the definition of a $\sum^{-1}\sum$-homomorphism slightly in order to allow homomorphisms which map a restricted portion of a decomposition algebra E to a composition algebra T. Let K be a relation on E_s. A <u>K-restricted</u> $\sum^{-1}\sum$ <u>homomorphism</u> from E to T is an S-indexed family of functions

$H = \langle h_S \rangle_{S \in S}$ such that

1) h_S: $E_S \rightarrow T_S$ for each $s \in S$ and

2) for each $x \in E_{\hat{S}}$ such that $K{:}x$, $h_{\hat{S}}{:}x = \sigma_T \cdot h^W \cdot \sigma_E{:}x$.

When $K{:}x$ is Defined$\cdot\sigma_E{:}x$ then we get back the original definition of a $\Sigma^{-1}\Sigma$-homomorphism.

2.5 Derived Preconditions

The synthesis method described later consists of a sequence of tasks many of which are carried out by a kind of deductive engine described in this section. Only those aspects of the engine needed for our synthesis examples are presented here. More details may be found in [20].

2.5.1 The Precondition Problem

The traditional problem of deduction has been to fknd a proof of a given formula in some theory. A more general problem, which we call the precondition problem, is most simply stated in the propositional calculus: given a goal A and hypothesis H, find a formula P, called a precondition, such that $P \wedge H \Rightarrow A$ is a tautology. In other words P provides any additional premises under which A can be shown to follow from H.

In this paper we derive preconditions in a many-sorted first-order theory Ψ. The data types, functions and predicates of Ψ needed for our examples are listed informally in the Appendix. The notions of term, atomic formula, literal and (well-formed) formula have their usual meaning [15]. We make use of a distinguished subset of the theorems of Ψ called known theorems which are assumed to be immediately available to the deductive system. The set of known theorems may change over time but initially includes all axioms of Ψ. All of the known theorems required by the examples are listed in the Appendix.

We introduce the notion of a precondition in a first-order logic by an example. Consider the following formulas

$$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i^2 \leq j^2] \tag{2.5.1}$$

$$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i = 0 \Rightarrow i^2 \leq j^2] \tag{2.5.2}$$

The first is invalid, the second valid. All that we have done in (2.5.2) is insert a sufficient condition, i=0, on the matrix $i^2 \leq j^2$ in (2.5.1). We call "i=0" an {i}-precondition of (2.5.1) because it contains only the variable i, and when we use it as we did in (2.5.2) we obtain a valid statement.

Formally, let $Q_1x_1\ Q_2x_2 \cdots Q_nx_n\ G$ be a closed formula not necessarily in prenex form where Q_i is either $\exists$ or $\forall$ for i=1,2,...,n. A $\{x_1x_2 \cdots x_n\}$-precondition of $Q_1x_1\ Q_2x_2 \cdots Q_nx_n\ G$ is a quantifier-free formula P dependent only on variables $x_1, x_2, \ldots, x_n$ such that

$$Q_1x_1Q_2x_2 \cdots Q_nx_n[\ P \Rightarrow G\ ]$$

is valid in Ψ. P is also a weakest $\{x_1x_2 \cdots x_n\}$-precondition if

$$Q_1x_1Q_2x_2 \cdots Q_nx_n[\ P \Leftrightarrow G\ ]$$

is valid in Ψ.

Example 2.5.1: Consider the formula $\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i^2 \leq j^2\ ]$

a) FALSE is a {}-precondition of (2.5.1) since
$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [\text{FALSE} \Rightarrow i^2 \leq j^2]$ is valid in Ψ,

b) i=0 is a {i}-precondition of (2.5.1) since
$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i=0 \Rightarrow i^2 \leq j^2]$ is valid in Ψ.

c) $i \leq j$ is a {i,j}-precondition of (2.5.1) since
$\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i \leq j \Rightarrow i^2 \leq j^2]$ is valid in Ψ.

Furthermore, note that each of the above preconditions are in fact weakest preconditions since the implication signs can each be replaced by equivalence signs without affecting validity. Note also that for any goal formula and set of variables the constant FALSE is a precondition.

In general a given goal may have many preconditions. Characteristics of a useful precondition seem to depend on the application domain. In program synthesis we want preconditions which are a) easily computable, b) in as simple a form as possible, and c) as weak as possible. (Criterion (c) prevents the boolean constant FALSE from being an acceptable precondition for all goals.) Clearly there is a tradeoff between these criteria. We currently measure each criterion by a separate heuristic function, then combine the results to form a net complexity measure on preconditions. We assume that such a complexity measure ranges over a well-founded set (such as $\mathbb{N}$ under the usual > relation) and

that we seek to minimize complexity over all preconditions.

Example 2.5.2: Consider again the formula $\forall i \in \mathbb{N}\ \forall j \in \mathbb{N}\ [i^2 \leq j^2]$ for which we want a useful $\{i,j\}$-precondition. Three candidates come to mind: FALSE, $i^2 \leq j^2$, and $i \leq j$. FALSE is certainly simple in form and semantics but it is not weak. Both $i^2 \leq j^2$ and $i \leq j$ are weakest preconditions however $i \leq j$ is the simpler of the two. Thus $i \leq j$ seems the most desireable.

The generality of the precondition problem allows us to define several well-known problems as special cases. The formula simplification problem involves transforming a given formula into an equivalent but simpler form. Formula simplification can be viewed as the problem of finding a weakest $\{x_1,\ldots x_n\}$-precondition of a given formula $Q_1x_1\ldots Q_nx_nG$. For example in the previous example we found $i \leq j$ to be a result of simplifying $i^2 \leq j^2$.

Theorem proving is the problem of showing that a given formula is valid in a theory by finding a proof of the formula. In terms of preconditions, theorem proving is the task of finding a weakest $\{\}$-precondition of a given formula. A precondition in no variables is one of the two propositional constants TRUE or FALSE. If we show that

$$\exists i \in \mathbb{N} \forall j \in \mathbb{N} [\text{TRUE} \Leftrightarrow i^2 \leq j^2]$$

is valid in Φ then we also have shown that

$$\exists i \in \mathbb{N} \forall j \in \mathbb{N} [i^2 \leq j^2]$$

is valid in Φ.

Formula simplification and theorem proving are opposite extremes in the spectrum of uses of preconditions since one involves finding a weakest precondition in all variables, and the other involves finding a weakest precondition in no variables. Between these extremes lies a use of preconditions which is critical to the synthesis method described later. Suppose that we wish to establish a relationship

$$\forall x_1\ \forall x_2\ \forall x_3\ \forall x_4\ [A{:}\langle x_1,x_2\rangle \wedge B{:}\langle x_2,x_3\rangle \wedge C{:}\langle x_3,x_4\rangle \Rightarrow D{:}\langle x_1,x_4\rangle] \quad (2.5.3)$$

where B, C, and D are known, but A is unknown and needs to be determined. Any $\{x_1,x_2\}$-precondition of (2.5.3) can be used for A. To see this let $A'{:}\langle x_1,x_2\rangle$ be any such precondition so

$$\forall x_1\ \forall x_2\ \forall x_3\ \forall x_4\ [A'{:}\langle x_1,x_2\rangle \Rightarrow (B{:}\langle x_2,x_3\rangle \wedge C{:}\langle x_3,x_4\rangle \Rightarrow D{:}\langle x_1,x_4\rangle)]$$

is valid. But this is equivalent to

$$\forall x_1\ \forall x_2\ \forall x_3\ \forall x_4\ [A':\langle x_1,x_2\rangle \wedge B:\langle x_2,x_3\rangle \wedge C:\langle x_3,x_4\rangle \Rightarrow D:\langle x_1,x_4\rangle]$$

In this example precondition derivation is similar to solving a linear equation in which all but one of the variables have been given values. A relation analogous to (2.5.3) must be established among the operators of a simple divide and conquer algorithm (the "separability condition" of Theorem 1 in Section 3.2). When all but one of the operators are known we use preconditions in order to derive the output condition of the unknown operator.

While we've shown that the precondition problem in a sense is more general than that of theorem proving we will see in the next section that actually deriving preconditions is much like a theorem proving process. The crucial difference is that in deriving a precondition P for a goal G we end up proving the validity of a formula involving P and G but we did not know ahead of time what we were going to prove! The proof process itself provides some of the premises of the formula which is finally proved valid.

2.5.2 A Formal System for Deriving Preconditions

Goal Preparation

In presenting a set of rules which allow us to derive preconditions we use the notation $\overset{A}{H}$ as an abbreviation of the formula

$$h_1 \wedge h_2 \wedge \ldots \wedge h_k \Rightarrow A$$

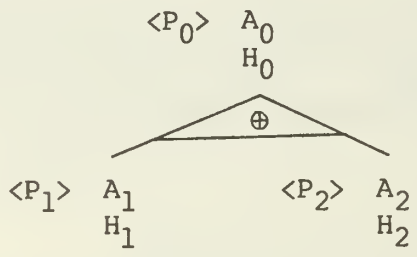
where $H = \{h_1,h_2,\ldots,h_k\}$. A goal statement $\overset{A}{H}$ and the known theorems of Ψ are prepared as follows. First, all occurences of equivalence ($\Leftrightarrow$) and implication ($\Rightarrow$) signs are eliminated and negation signs are moved in as far as possible. H and the known theorems of Ψ are then skolemized in the usual way [14], i.e., existentially quantified variables are replaced by skolem functions of the universally quantified variables on which they depend. Quantifiers are then dropped with the understanding that all remaining variables are universally quantified. The goal A is skolemized in a dual manner with universally quantified variables replaced by skolem functions of the existential variables on which they depend. All quantifiers are then dropped with the understanding that all variables in A which remain are existentially quantified. The preparation of A is equivalent (via duality of goals and assertions) to preparing $\sim$A as an

hypothesis then taking the negation of the result as our prepared goal.

All of the derivations treated in this paper involve only universally quantified variables. Consequently during goal preparation each variable in the goal is replaced by a skolem function of no arguments (i.e. a constant). Rather than invent a special notation for these skolem constants we will simply use the variable name itself. Thus in example derivations symbols for variables, such as "x", should be regarded as skolem constants.

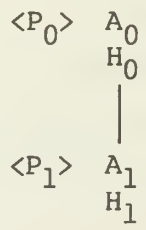
Reduction Rules

Rules which reduce a goal statement to two subgoal statements are expressed in the following form:

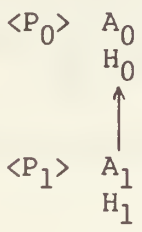


where A_0,A_1, and A_2 are goal formulas, H_0, H_1, and H_2 are sets of hypotheses, P_0, P_1, and P_2 are formulas (the derived preconditions), and $\oplus$ is either $\vee$ or $\wedge$. A rule of this form asserts that if P_i is a (weakest) precondition of $\begin{matrix}A_i\\H_i\end{matrix}$ where i=1 or 2 then P_0 is a (weakest) precondition of $\begin{matrix}A_0\\H_0\end{matrix}$. P_0 generally is $P_1 \oplus P_2$. Typically a deductive process also returns a substitution for any variables in the goal. Substitutions do not play an important role in the examples of this paper so for simplicity we omit them whenever possible. They are fully treated in [20].

Rules which reduce a goal statement to one subgoal are notated



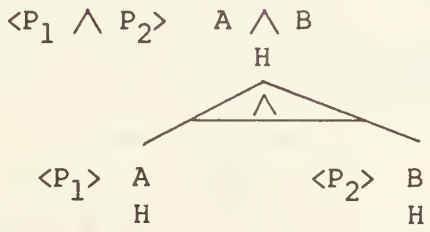
Occasionally, as in the application of known theorems which are implications, the relation between goal and subgoals is not one of equivalence but implication. Rules of this kind are notated



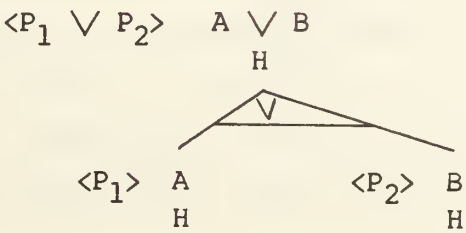
which asserts that if P_1 is a precondition of $\begin{matrix}A_1\\H_1\end{matrix}$ then P_0 is a precondition of $\begin{matrix}A_0\\H_0\end{matrix}$. For rules of this kind we cannot assert that P_0 is a weakest precondition of $\begin{matrix}A_0\\H_0\end{matrix}$ even if P_1 is known to be a weakest precondition of $\begin{matrix}A_1\\H_1\end{matrix}$.

The following rules are for the most part extensions of typical goal reduction rules [5,14].

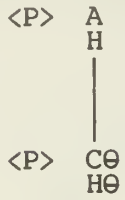
R1. Reduction of Conjunctive Goals



R2. Reduction of Disjunctive Goals

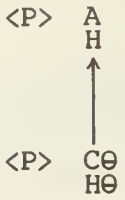


R3. Application of an Equivalence Theorem



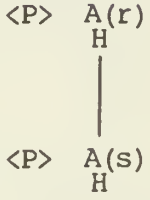
if $C \Leftrightarrow B$ is a known theorem of Ψ
or an hypothesis in H and Θ unifies {A,B}

R4. Application of an Implicational Theorem



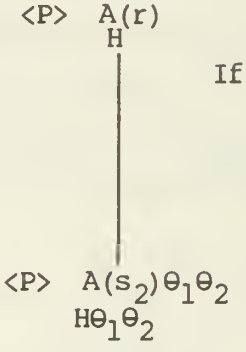
if $C \Rightarrow B$ is a known theorem of Ψ
or hypothesis in H, where Θ unifies {A,B}

R5. Substitution of Equal Terms



if r=s is an hypothesis in H
or a known theorem of Ψ

R6. Conditional Equality Substitution



If $B \Rightarrow s_1 = s_2$ is a known theorem
where Θ_1 unifies $\{r,s_1\}$ and
Θ_2 unifies $B\Theta_1$ with a
hypothesis or known theorem.

Primitive Rules

A reduction rule generates a precondition for a goal by decomposing it into subgoals, then composing the derived preconditions of the subgoals. We also use two rules, called primitive rules, which can directly generate a precondition

for a goal. Both are described by notations of the form $\langle P \rangle \ {}^{A}_{H}$ which assert that P is a precondition of $^{A}_{H}$ if the associated condition holds.

P1. $\langle TRUE \rangle \ {}^{A}_{H}$ if A can be directly evaluated to TRUE, or if Θ unifies $\{A,B\}$ where B is a known theorem of Ψ or $B \in H$.

P2. $\langle H' \Rightarrow A' \rangle \ {}^{A}_{H}$ if we seek a $\{x_1,\ldots,x_n\}$-precondition and A' depends only on the variables $x_1,\ldots,x_n$ and H' has the form $\bigwedge_{j=1}^{n} h_{i_j}$ where $H = \{h_1,h_2,\ldots,h_k\}$ and $\{h_{i_j}\}_{j=1,m} \subseteq H$ and for each j, $1 \leq j \leq m$, h_{i_j} depends only on the variables $x_1,x_2,\ldots,x_n$.

The primitive rule P1 always generates weakest preconditions but P2 does not in general unless A' is A and H' is H.

The Deduction Process

The derivation of a precondition of goal statement $^{A}_{H}$ can be described by a two stage process. In the first phase reduction rules are repeatedly applied to goals reducing them to subgoals. Primitive rule P1 is applied whenever possible. If no reduction rules can be applied to a goal (or if we simply desire to cut short the deduction) primitive rule P2 is applied. The result of this reduction process is a goal tree in which 1) nodes represent goals/subgoals, 2) arcs represent reduction rule applications, and 3) leaf nodes represent goals to which a primitive rule has been applied.

The second phase involves the bottom-up composition of preconditions. Initially each application of a primitive rule to a goal yields a precondition. Subsequently whenever a precondition has been found for each subgoal of a goal $^{A}_{H}$ then a precondition is composed for $^{A}_{H}$ according to the reduction rule employed. Each newly composed precondition is then run through a simplification process.

Usually several reduction rules can be applied to a given goal and each rule will generate a precondition. We make use of a complexity measuring function to select that precondition of least complexity among the alternatives.

Example 2.5.3: Suppose that we wish to derive a $\{x_0,x_1,x_2\}$-precondition of

$\forall \langle x_0,x_1,x_2\rangle \in LIST(\mathbb{N}) \times LIST(\mathbb{N}) \times LIST(\mathbb{N})$
$\forall \langle z_0,z_1,z_2\rangle \in LIST(\mathbb{N}) \times LIST(\mathbb{N}) \times LIST(\mathbb{N})$
$[Bag{:}x_1 = Bag{:}z_1 \wedge Ordered{:}z_1 \wedge Bag{:}x_2 = Bag{:}z_2 \wedge Ordered{:}z_2 \wedge$
$Append{:}\langle z_1,z_2\rangle = z_0 \Rightarrow Ordered{:}z_0]$.

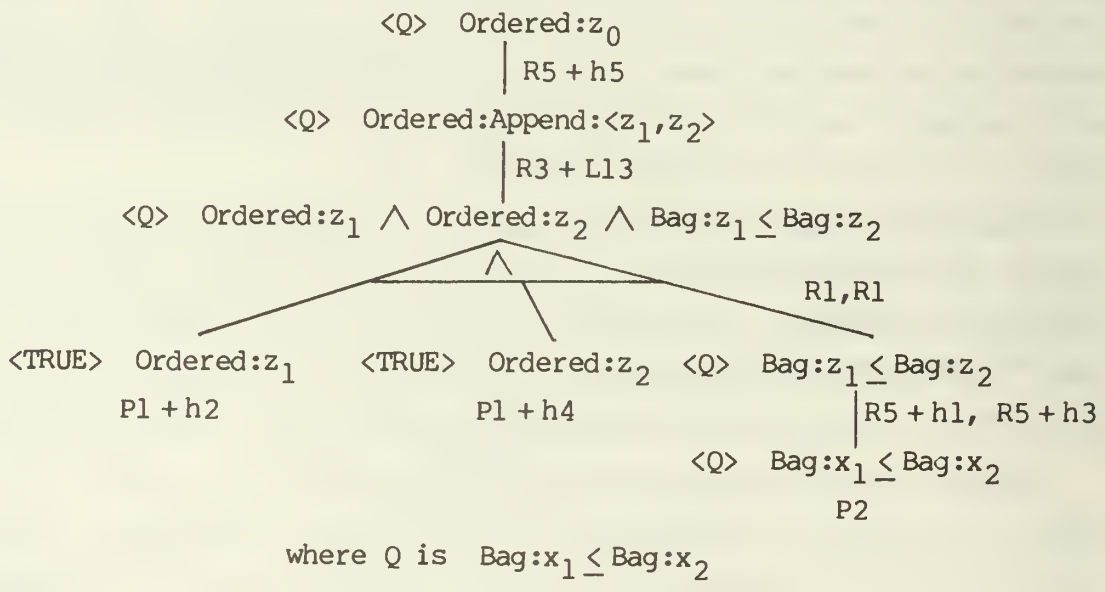
This precondition problem is taken from the synthesis of a Quicksort algorithm in Section 4.3.3. A goal tree representing a formal derivation of the precondition $Bag{:}x_1 \leq Bag{:}x_2$ is given in Figure 6. In this example and all that follow we annotate the arcs of goal trees with the name of the rule and known theorem or hypothesis used and note the primitive rule used on each leaf node. In this

---

Hypotheses: h1. $Bag{:}x_1 = Bag{:}z_1$
h2. $Ordered{:}z_1$
h3. $Bag{:}x_2 = Bag{:}z_2$
h4. $Ordered{:}z_2$
h5. $Append{:}\langle z_1,z_2\rangle = z_0$

Variables: $\{x_0,x_1,x_2\}$

Goal 1:



where Q is $Bag{:}x_1 \leq Bag{:}x_2$

Figure 6: Example Derivation of a Precondition

---

example the given goal Ordered:z_0 is reduced by application of the rule R5 (equality substitution) together with hypothesis h5. The resulting subgoal Ordered:Append:$\langle z_1, z_2 \rangle$ is further reduced by rule R3 (application of equivalence theorems) together with the known theorem

$$\text{Ordered}:x_1 \wedge \text{Ordered}:x_2 \wedge x_1 \leq x_2 \Leftrightarrow \text{Ordered} \cdot \text{Append}:\langle x_1, x_2 \rangle .$$

(called L13) to the subgoal

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \wedge \text{Bag}:z_1 \leq \text{Bag}:z_2 .$$

This conjunction is decomposed by two applications of rule R1 (reduction of conjunctions) into the three subgoals on the fourth line. The primitive rule P1 matchs the first subgoal Ordered:z_1 with hypothesis h2, generating precondition TRUE. Similarly, P1 matchs the second subgoal Ordered:z_2 with hypothesis h4, generating precondition TRUE. The third subgoal Bag:$z_1 \leq$ Bag:z_2 is reduced by the application of rule R5 twice with hypotheses h1 and h3 yielding subgoal Bag:$x_1 \leq$ Bag:x_2. This subgoal depends only on the variables x_1 and x_2 and we can apply primitive rule P2 yielding the precondition Bag:$x_1 \leq$ Bag:x_2 (which we call Q for brevity). In the composition phase of the derivation the preconditions generated by the primitive rules are passed up the goal tree and composed. The composed precondition of the subgoal

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \wedge \text{Bag}:z_1 \leq \text{Bag}:z_2$$

is in fact

$$\text{TRUE} \wedge \text{TRUE} \wedge \text{Bag}:x_1 \leq \text{Bag}:x_2$$

which simplifies to Bag:$x_1 \leq$ Bag:x_2. In this example and the sequel we record only the simplified form of a composed precondition. Finally Bag:$x_1 \leq$ Bag:x_2 is passed all the way back up the tree to become the derived precondition of the original goal.

3. Top-Down Program Synthesis

In Section 1 we discussed the notion of top-down design. We now describe the general structure of a system for the top-down design of algorithms. As depicted in Figure 6, the system takes as input a incomplete specification of a problem and generates as output an algorithm plus completed specification. The advantage of using incomplete specifications is threefold. First, the user need not be concerned with how to solve his/her problem but rather can focus on the nature and structure of the problem itself. Second, other than having the user supply a complete program, it is only with specifications that we are able to completely verify that the user's intentions have been met by a potential solution. Finally, incomplete specifications are easier to create since the user need not be concerned with supplying all necessary input conditions - the system will supply them automatically.

The programming knowledge needed for top-down program design is organized about the two central aspects of the design process: 1) how to directly

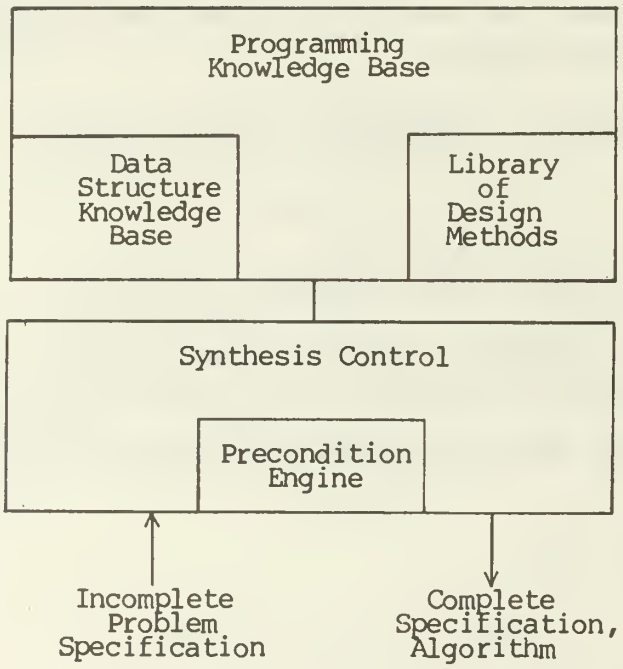


Figure 6. Structure of a Top-Down Program Synthesis System.

construct a solution to a so-called primitive problem, and 2) how to break a nonprimitive problem down into subproblems and assemble a solution based on solutions to subproblems. Knowledge about the first aspect is presented in Section 3.1. Broadly we envision knowledge of the second kind coming in the form of design theories for various classes of algorithms. A design theory consists of three parts:

1. A program schema, which is a parameterized program template with uninterpreted symbols for subprograms. The program schema characterizes the structure common to algorithms in the class.

2. A formula providing sufficient conditions for the total correctness of the schema with respect to a generic problem specification. Necessarily these conditions are formula schemas containing uninterpreted symbols for the specifications of subprograms in the program schema, and for predicates in the problem specification. Note that they link a problem specification and a program schema.

3. A design method which attempts to instantiate the schema in order to satisfy a given problem specification. It works by deriving subproblem specifications in such a way that the sufficient conditions are satisfied. Loosely speaking, in our approach a design method uses the sufficient conditions to "solve for" subproblem specifications.

The main result of this paper is a design theory for the class of simple divide and conquer algorithms.

The Programming Knowledge Base in Figure 6 consists of two parts. The Data Structure Knowledge Base stores all system knowledge about data types, their operators, and their properties. It is discussed in Section 3.1. The Library of Design Theories is a collection of design theories as discussed above. A program schema for the class of simple divide and conquer algorithms and sufficient conditions for correctness of the schema are presented in Section 3.2. In Section 3.3 we present our design method for simple divide and conquer algorithms. Several of our examples require the synthesis of simple conditional programs. A collection of design methods for conditional programs will be treated elsewhere, but assumed as given for our present examples.

The Synthesis Control Module controls the top-down design process. Among its tasks are obtaining specifications from the user, selecting and applying design theories, and managing the resulting tree structure. Included in the

Control Module is a precondition engine as discussed in Section 2.5.

We are currently constructing a system with the structure of Figure 6. A precondition engine has been built which can handle most of the derivations given in the example syntheses below.

3.1 Data Structure Knowledge Base

The data structure knowledge base (DSKB) is the repository of all system knowledge about data types, their functions, algebras, and properties. The data types represented in the DSKB, called known data types, may change over time but initially include the primitive types of the target programming language. The functions of the DSKB, called known functions, also may change over time under user definition but initially include the primitive functions of the target programming language. The algebras in the DSKB, called known algebras, may also change over time but initially include at least one constructive algebra for each known data type. Logical statements involving the known data types and functions, called known theorems, also may change over time as new theorems are proved or added by a user but initially include the axioms which describe the primitive data types and functions of the programing language. The DSKB assumed for the purposes of this paper is presented in the Appendix.

The organization (and structure) of the DSKB depends on the various roles it plays in the synthesis system. The DSKB is used as the lexicon of the specification language. We allow in specifications any formula constructable from the known types and functions. As the DSKB changes over time so does the specification language. From the specification language point of view the DSKB defines a first-order language.

The precondition engine uses the DSKB as a knowledge base of theorems to use during its derivations. From this point of view the DSKB is a partial representation of a first-order theory (partial in the sense that only the known theorems are explicitly represented).

As previously indicated the synthesis method for divide and conquer programs involves the construction of algebras on various domains. Thus organization of the data types and functions, relations, and theorems along the lines of algebras and subalgebras may be useful.

Although we do not include it here, ideally the DSKB also requires considerable programming knowledge of the kind described by Barstow in [3]. The reason is this: the synthesis method constructs a program out of known operators and relations mapping the input type to the output type given in the specification. If the input and/or output types or operators or relations are not primitive then the constructed algorithm must be further refined into target language primitives. Barstow has explored the kind of knowledge and processing required to perform this refinement process.

Matching Known Functions Against a Given Specification

The top-down decomposition process terminates in specifications which can be satisfied by known functions. Consequently a basic operation of the DSKB is to retrieve any known functions satisfying a given specification. The following two theorems provide the basis for two variants of this operation. Proposition 3 suggests a matching operation which is useful when we have a given specification and we wish to see if any of a library of functions, each described by a specification, satisfy it. Proposition 4 is useful when a known function is described not by specification but by axioms (known theorems).

Proposition 3: Let $\overline{\Pi}_1 = \langle D_1, R_1, I_1, O_1 \rangle$ and $\overline{\Pi}_2 = \langle D_2, R_2, I_2, O_2 \rangle$ be specifications. If

(a) $D_2 \subseteq D_1$

(b) $R_1 \subseteq R_2$

(c) J is an $\{x\}$-precondition of $\forall x \in D_2\ [I_2{:}x \Rightarrow I_1{:}x]$,

(d) K is an $\{x\}$-precondition of

$$\forall x \in D_2\ \forall z \in S_1\ [I_2{:}x \wedge O_1{:}\langle x,z\rangle \Rightarrow O_2{:}\langle x,z\rangle]$$

then any function satisfying $\overline{\Pi}_1$ also satisfies $\overline{\Pi}_2$ with derived input condition $J \wedge K$.

Proof: Let F be any function satisfying $\overline{\Pi}_1$, thus

$$\forall x \in D_1\ [I_1{:}x \Rightarrow O_1{:}\langle x, F{:}x\rangle].$$

We must show

$$\forall x \in D_2 [I_2{:}x \wedge J{:}x \wedge K{:}x \Rightarrow O_2{:}\langle x, F{:}x\rangle]$$

where J and K are preconditions satisfying conditions (c) and (d) respectively.

Let $x \in D_2$ and assume $I_2{:}x \wedge J{:}x \wedge K{:}x$. By conditions (a) and (c) we can infer $I_1{:}x$. Since F satisfies $\overline{\Pi}_1$ we obtain $O_1{:}\langle x,F{:}x\rangle$. Since $F{:}x \in R_1$, and by condition (d) $K{:}x \wedge I_2{:}x \wedge O_1{:}\langle x,F{:}x\rangle \Rightarrow O_2{:}\langle x,F{:}x\rangle$, we obtain via modus ponens that $O_2{:}\langle x,F{:}x\rangle$. Since x was taken as an arbitrary element of D_2 it follows that

$$\forall x \in D_2\ [I_2{:}x \wedge J{:}x \wedge K{:}x \Rightarrow O_2{:}\langle x,F{:}x\rangle]$$

i.e. F satisfies $\overline{\Pi}_2$ with derived input condition $I_2 \wedge J \wedge K$. QED

Example 3.1.1: One of the known functions of the DSKB, called Listsplit, takes a list and splits it roughly in half. It is specified as follows:

$$\text{Listsplit}{:}x_0 = \langle x_1,x_2\rangle \text{ such that } x_0 = \text{Append}{:}\langle x_1,x_2\rangle \wedge$$
$$\text{Length}{:}x_1 = \text{Length}{:}x_0 \text{ div } 2 \wedge \text{Length}{:}x_2 = (1 + \text{Length}{:}x_0) \text{ div } 2$$
$$\text{where Listsplit}{:}\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}).$$

By x div k we mean integer division by k. Thus 5 div 2 = 2. During the synthesis of a mergesort algorithm we derive the following specification:

$$\text{Decompose}{:}y_0 = \langle y_1,y_2\rangle \text{ such that Length}{:}y_0 > \text{Length}{:}y_1 \wedge \text{Length}{:}y_0 > \text{Length}{:}y_2$$
$$\text{where Decompose}{:}\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}).$$

We can match Decompose and Listsplit using Proposition 3. Since the input domain, the output domain, and the input condition coincide it remains to derive a $\{y_0\}$-precondition of

$$\forall \langle y_0,y_1,y_2\rangle \in \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})\ [y_0 = \text{append}{:}\langle y_1,y_2\rangle \wedge$$
$$\text{Length}{:}y_1 = \text{Length}{:}y_0 \text{ div } 2 \wedge \text{Length}{:}y_2 = (1 + \text{Length}{:}y_0) \text{ div } 2$$
$$\Rightarrow \text{Length}{:}y_0 > \text{Length}{:}y_1 \wedge \text{Length}{:}y_0 > \text{Length}{:}y_2].$$

We have created the precondition problem by making the following instantiations into the condition (d) of Proposition 3:

LIST(IN) replaces D_1,R_1,D_2,R_2

TRUE replaces I_1,I_2

the output condition of Listsplit replaces O_1, and

the output condition of Decompose replaces O_2.

In Figure 7, we derive the precondition $\text{Length}{:}y_0 > 0 \wedge \text{Length}{:}y_0 > 1$ which simplifies to $\text{Length}{:}y_0 > 1$. Thus according to Proposition 3 Listsplit satisfies

the specification of Decompose with derived input condition Length:y_0>1. This means that we can use the function Listsplit for the problem Decompose provided that it is never passed an argument of length zero or one.

Hypotheses: 1. x_0 = append:<x_1,x_2>
2. Length:x_1 = Length:x_0 div 2
3. Length:x_2 = (1 + Length:x_0) div 2

Variables: {x_0}

Goal 1:

<Q1> Length:x_0 > Length:x_1
| R5 + h2
<Q1> Length:x_0 > Length:x_0 div 2
↑ R4 + n2
<Q1> Length:x_0 + Length:x_0 > Length:x_0
| R3 + n1
<Q1> Length:x_0 > 0
P2

where Q1 is Length:x_0>0

Goal 2:

<Q2> Length:x_0 > Length:x_2
| R5 + h3
<Q2> Length:x_0 > (1 + Length:x_0) div 2
↑ R4 + n2
<Q2> Length:x_0 + Length:x_0 > 1 + Length:x_0
| R3 + n1
<Q2> Length:x_0 > 1
P2

where Q2 is Length:x_0>1

Figure 7: Matching the specification of Decompose with the specification of Listsplit

---

Proposition 4: Let F be a function with domain D_1 and codomain R_1 and let $\Pi_2 = \langle D_2, R_2, I_2, O_2 \rangle$ be a specification. If

(a) $\forall x \in D_1\ [I_1{:}x \Rightarrow \text{Defined}\cdot F{:}x]$

(b) $D_2 \subseteq D_1$

(c) $R_1 \subseteq R_2$

(d) J is an $\{x\}$-precondition of $\forall x \in D_2\ [I_2{:}x \Rightarrow I_1{:}x]$

(e) K is an $\{x\}$-precondition of

$$\forall x \in D_2\ \forall z \in R_1 [J{:}x \wedge I_2{:}x \wedge z = F{:}x \Rightarrow O_2{:}\langle x,z\rangle]$$

then F satisfies Π_2 with derived input condition $J \wedge K$.

Proof: We must show

$$\forall x \in D_2 [I_2{:}x \wedge J{:}x \wedge K{:}x \Rightarrow O_2{:}\langle x, F{:}x\rangle].$$

Let x be an arbitrary element of D_2 and assume $I_2{:}x \wedge J{:}x \wedge K{:}x$. By condition (d) we can infer $I_1{:}x$. Since $D_2 \subseteq D_1$ we have $x \in D_1$ and by condition (a) we can infer $\text{Defined}\cdot F{:}x$. $F{:}x$ is in R_2 since $F{:}x \in R_1 \subseteq R_2$ by condition (c). Finally by condition (e) we can infer $O_2{:}\langle x, F{:}x\rangle$. QED

Proposition 4 is used when we do not have a specification for an operator F but its behavior is fully described by the known theorems of the DSKB. Such a situation arises for certain primitive functions of the target language. These are used to specify and define other functions but cannot themselves be described in terms of more primitive functions. Their behavior in relation to other primitive functions is instead described by axioms (known theorems). All that is required for the matching operation suggested by Proposition 4 is 1) the domain and codomain of F, 2) conditions under which F is defined, and 3) axiomatic specification of its behavior.

Example 3.1.2: The operator Cons is described by its interaction with other operators such as First, Rest, and Bag. Letting x vary over LIST(IN) and a over IN, the usual axioms include:

1. $\text{Cons}\cdot[\text{First}, \text{Rest}]{:}x = x$
2. $\text{First}\cdot\text{Cons}{:}\langle a,x\rangle = a$
3. $\text{Rest}\cdot\text{Cons}{:}\langle a,x\rangle = x$

4. Defined·Cons:<a,x>

5. Bag·Cons:<a,x> = Add:<a,Bag:x>.

We can use Proposition 4 to show that Cons satisfies the following specification:

$$\overline{\Pi}_2:\langle a,x\rangle = z \text{ such that } Add:\langle a,Bag:x\rangle = Bag:z$$
$$\text{where } \overline{\Pi}_2: \mathbb{N} \times LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}).$$

The input and output domains of $\overline{\Pi}_2$ and Cons coincide. By axiom 4 we implicitly have TRUE as input condition for Cons, therefore we can derive TRUE for J in condition (d) of the theorem. Finally according to condition (e) we attempt to find an {a,x}-precondition of

$$\forall \langle a,x\rangle \in \mathbb{N} \times LIST(\mathbb{N})\ [TRUE \wedge TRUE \Rightarrow Add:\langle a,Bag:x\rangle = Bag:Cons:\langle a,x\rangle].$$

This formula is easily shown valid by using axiom 5 above. Thus by Proposition 4 we conclude that Cons satisfies $\overline{\Pi}_2$ with derived input condition TRUE.

It may be that no single known function satisfies a given specification but that a structure of known functions will satisfy it. If the specification requests a mapping of type $D_1 \times \ldots \times D_m \rightarrow R_1 \times \ldots \times R_n$ then a function structure of the form $[f_1,f_2\ldots,f_n]$ is required. A straightforward backtracking algorithm for placing these n functions can be used to find all potential structures. Once a structure is found with the correct input and output types the matching operation can be invoked to verify satisfaction of the specification.

Example 3.1.3: Consider the specification

$$\overline{\Pi}:x = \langle a,z\rangle \text{ such that } x = Cons:\langle a,z\rangle$$
$$\text{where } \overline{\Pi}:LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N}).$$

Here a function structure of the form $[f_1,f_2]$ is required. For f_1 we might consider all known functions mapping LIST(ℕ) to ℕ such as First, Length, Min, etc. For f_2 we might consider all mappings from LIST(ℕ) to LIST(ℕ) such as Rest, Sort, etc. Passing [First,Rest] to the matching operations described in the previous section we can derive $x \neq nil$ as derived input condition. The derived input conditions for the other potential structures are not very weak so they are discarded.

3.2 Problem Reduction Representations and Simple Divide and Conquer Algorithms

A problem specification typically provides few indications of how to go about solving it. Rather than attempt to map directly from the specification to a satisfying program, our approach to synthesizing a divide and conquer algorithm proceeds by enriching the specification with additional structure in the form of algebras on the input and output domains. The construction of these algebras is constrained not only by each other but also by the input and output conditions. The result of this enrichment process is called a problem reduction representation of the original specification. A program may then be straightforwardly constructed from the components of the representation.

Let Σ be a simple S-sorted signature of type $\langle w,\hat{s}\rangle$ where $w \in S^*$, $w = w_1,w_2,\ldots,w_n$, and $\hat{s} \in S$. A Σ-problem reduction representation (Σ-PRR) is a system $\hat{\Pi} = \langle E,T,J,P\rangle$
where

1. E is a Σ^{-1}-algebra called the input algebra
2. T is a Σ-algebra called the output algebra
3. $J = \langle J_s\rangle_{s \in S}$ is an S-indexed family of relations on E (i.e. $J_s \subseteq E_s$ for $s \in S$) called the family of input conditions
4. $P = \langle P_s\rangle_{s \in S}$ is an S-indexed family of relations on $E \times T$ (i.e. $P_s \subseteq E_s \times T_s$ for each s) called the family of output conditions.

For each $s \in S$ let $\hat{\Pi}_s$, called a component problem, denote the problem specification $\langle E_s,T_s,J_s,P_s\rangle$. $\hat{\Pi}_{\hat{s}}$ will be called the principal problem and for each $s \in S-\hat{s}$ $\hat{\Pi}_s$ will be called an auxiliary problem. $\hat{\Pi}$ represents specification $\Pi = \langle D,R,I,O\rangle$ if $\hat{\Pi}_{\hat{s}} = \Pi$.

An S-indexed family of functions $F = \langle f_s\rangle_{s \in S}$ satisfies Σ-problem reduction representation $\hat{\Pi} = \langle E,T,J,P\rangle$ if

1. for each $s \in S$, f_s satisfies $\hat{\Pi}_s = \langle E_s,T_s,J_s,P_s\rangle$, and
2. F is a L-restricted homomorphism from E to T for some relation L on $E_{\hat{s}}$.

Clearly if $F = \langle f_s\rangle_{s \in S}$ satisfies $\hat{\Pi}$ and $\hat{\Pi}$ represents Π then $f_{\hat{s}}$ satisfies Π.

An S-indexed family of functions $F = \langle f_s\rangle_{s \in S}$ is called a simple divide and conquer program if $f_{\hat{s}}$ has the form

$$f_{\hat{s}}:x \equiv \text{if}$$
$$q:x \rightarrow h:x[]$$
$$\neg q:x \rightarrow \sigma_T \cdot f^w \cdot \sigma_E:x$$
$$\text{fi}.$$

We call σ_E the decomposition operator, σ_T the composition operator, f_s an auxiliary operator for each $s \in S-\hat{s}$, and h the primitive operator. The term "simple" is used to denote the restriction that $f_{\hat{s}}$ employs a single pair of composition and decomposition operators. Our main synthesis task is to construct a Σ-PRR which represents a given specification then construct a simple divide and conquer program which satisfies it.

Theorem 1 characterizes programs which satisfy a Σ-PRR and provides the basis for the synthesis method presented in Section 3. Most of the conditions of Theorem 1 are straightforward requirements on the correctness of the programs f_s for each $s \in S$. An exception, and perhaps the most interesting condition, is the "separability" condition. In words it states that if input x_0 decomposes into subinputs $x_1,\ldots,x_n$, and $z_1, \ldots,z_n$ are feasible outputs with respect to these subinputs respectively, and $z_1,\ldots,z_n$ compose to form z_0 then z_0 is a feasible solution to input x_0. Loosely put: feasible outputs compose to form feasible outputs. It is the principal link between the algebras E and T, and the input and output conditions.

Theorem 1: (Sufficient conditions for the existence of a simple divide and conquer solution to a problem reduction representation) Let $\hat{\Pi} = \langle E,T,J,P\rangle$ be a Σ-PRR where Σ is a simple S-sorted signature of type $\langle w,\hat{s}\rangle$, and let $F = \langle f_s\rangle_{s \in S}$ be an S-sorted family of functions. Let $\succ$ be a well-founded ordering on $E_{\hat{s}}$ and let O_E and O_T be relations on $E^{\hat{s}w}$ and $T^{\hat{s}w}$ respectively. If

(1) (Specification of σ_E) the decomposition operator σ_E satisfies the specification

$$\sigma_E:x_0 = \langle x_1,\ldots,x_n\rangle \text{ such that } J_{\hat{s}}:x_0 \Rightarrow \bigwedge_{i \in \underline{n}} (J_{w_i}:x_i \wedge (i = \hat{s} \Rightarrow x_0 \succ x_i)) \wedge O_E:\langle x_1,\ldots,x_n\rangle$$
$$\text{where } \sigma_E:E_{\hat{s}} \rightarrow E^w$$

with derived input condition K_E;

(2) (Specification of σ_T) the composition operator σ_T satisfies the specification

$$\sigma_T:\langle z_1,\ldots,z_n\rangle = z_0 \text{ such that } O_T:\langle z_0,z_1,\ldots,z_n\rangle$$
$$\text{where } \sigma_T:T^W \rightarrow T_{\hat{S}}$$

with derived input condition TRUE;

(3) (Separability of P) the following formula is valid:

$$\forall\langle x_0,x_1,\ldots,x_n\rangle \in E^{\hat{S}W}\ \forall\langle z_0,z_1,\ldots,z_n\rangle \in T^{\hat{S}W}$$
$$[\sigma_E:x_0 = \langle x_1,\ldots,x_n\rangle \wedge \bigwedge_{i\in\underline{n}} P_{w_i}:\langle x_i,z_i\rangle \wedge \sigma_T:\langle z_1,\ldots,z_n\rangle = z_0 \Rightarrow P_{\hat{S}}:\langle x_0,z_0\rangle]$$

(4) (Solutions to Auxiliary Problems) for each $s \in S-\hat{S}$, f_s satisfies $\overline{\Pi}_s = \langle E_s,T_s,J_s,P_s\rangle$;

(5) (Structure of $f_{\hat{S}}$)

(5.1) $f_{\hat{S}}:x \equiv$ if

$$q:x \rightarrow h:x[]$$
$$\neg q:x \rightarrow \sigma_T\cdot f^W\cdot\sigma_E:x$$

fi

(5.2) the guard $\neg q$ is K_E,

(5.3) h satisfies the specification $\langle E_{\hat{S}}, T_{\hat{S}}, J_{\hat{S}}\wedge q, P_{\hat{S}}\rangle$,

(5.4) the following formula is valid:

$$\forall x \in E_{\hat{S}}\ [J_{\hat{S}}:x \Rightarrow \text{Defined}\cdot q:x];$$

then the simple divide and conquer program F satisfies $\hat{\overline{\Pi}}$.

Proof: To show that F satisfies $\hat{\overline{\Pi}}$ we first show that f_s satisfies $\hat{\overline{\Pi}}_s$ for each $s \in S$. By condition (4) this is so for each $s \in S-\hat{S}$. To show that $f_{\hat{S}}$ satisfies $\overline{\Pi}_{\hat{S}} = \langle E_{\hat{S}},T_{\hat{S}},J_{\hat{S}},P_{\hat{S}}\rangle$ we will prove

$$\forall x \in E_{\hat{S}}\ [J_{\hat{S}}:x \Rightarrow P:\langle x,f_{\hat{S}}:x\rangle]$$

by structural induction[1] on $E_{\hat{S}}$.

---

[1] Structural induction on a well-founded set $\langle W,\succ\rangle$ is a form of mathematical induction described by

$$\forall x \in W\ \forall y \in W[x \succ y \wedge Q:y \Rightarrow Q:x] \Rightarrow \forall x \in W\ Q:x$$

Let x be an object in $E_{\hat{s}}$ such that $J_{\hat{s}}:x$ holds and assume (inductively) that $J_{\hat{s}}:y \Rightarrow P:\langle y, f_{\hat{s}}:y\rangle$ holds for any $y \in E_{\hat{s}}$ such that $x \succ y$. From $J_{\hat{s}}:x$ and condition (5.4) it follows that $q:x$ is defined thus there are two cases to consider: $q:x = \text{TRUE}$ and $\sim q:x = \text{TRUE}$.

Case 1: Assume that $q:x = \text{TRUE}$ then $f_{\hat{s}}:x = h:x$ by construction of $f_{\hat{s}}$. Furthermore according to condition (5.3) we have $J_{\hat{s}}:x \wedge q:x \Rightarrow P_{\hat{s}}:\langle x, h:x\rangle$ from which we easily infer $P_{\hat{s}}:\langle x, h:x\rangle$ or equivalently $P_{\hat{s}}:\langle x, f_{\hat{s}}:x\rangle$.

Case 2: Assume that $\sim q:x = \text{TRUE}$ then $f_{\hat{s}}:x = \sigma_T \cdot f^w \cdot \sigma_E:x$. We will show that $P_{\hat{s}}:\langle x, f_{\hat{s}}:x\rangle$ by using the inductive assumption and modus ponens on the separability condition. By condition (5.2) $\sim q$ is K_E so $K_E:x = \text{TRUE}$. Since $J_{\hat{s}}:x$ also holds, and σ_E satisfies its specification in condition (1), the output condition of σ_E also holds. Let $\sigma_E:x = \langle x_1, \ldots, x_n\rangle$. We have for each $i \in \underline{n}$ $J_{w_i}:x_i$. Consider x_i for each $i \in \underline{n}$. If $w_i \neq \hat{s}$ then by condition (4) we have $J_{w_i}:x_i \Rightarrow P_{w_i}:\langle x_i, f_{w_i}:x_i\rangle$ and we obtain by modus ponens $P_{w_i}:\langle x_i, f_{w_i}:x_i\rangle$. If on the other hand $w_i = \hat{s}$ then by condition (1) we have $x_0 \succ x_i$ and thus by our inductive assumption $J_{w_i}:x_i \Rightarrow P_{w_i}:\langle x_i, f_{w_i}:x_i\rangle$. Again we obtain $P_{w_i}:\langle x_i, f_{w_i}:x_i\rangle$ by modus ponens. By condition (2) we have $O_T:\langle \sigma_T:\langle f_{w_1}:x_1, \ldots, f_{w_n}:x_n\rangle, z_1, \ldots, z_n\rangle$ where $\sigma_T:\langle f_{w_1}:x_1, \ldots, f_{w_n}:x_n\rangle = f_{\hat{s}}:x$. We have now established the antecedent of condition (3) enabling us to infer $P_{\hat{s}}:\langle x, f_{\hat{s}}:x\rangle$.

We have shown that for each $s \in S$ f_s satisfies $\hat{\Pi}_s = \langle E_s, T_s, J_s, P_s\rangle$. It remains to show that F is a L-restricted homomorphism from E to T for some relation L on $E_{\hat{s}}$. Let L be K_E. If $x \in E_{\hat{s}}$ and $K_E:x$ holds then by condition (5.2) $\sim q:x$ holds thus $f_{\hat{s}}:x = \sigma_T \cdot f^w \cdot \sigma_E:x$. QED

In overall structure the synthesis method systematically attempts to satisfy the conditions of Theorem 1 more or less in the stated order. Conditions (1), (2), and (3) are used to construct E and T. Condition (5.2) is used to determine q. Finally condition (5.3) is used to construct h. The various derived functions are assembled according to the schema in condition (5.1).

---

i.e., if Q:x can be shown to follow from the assumption that Q:y holds for each y such that $x \succ y$, then we can conclude that Q:x holds for all x.

3.3 Design Method for Simple Divide and Conquer Algorithms

The synthesis method described in this section constructs a simple divide and conquer algorithm along the lines suggested by Theorem 1. We first describe the method formally then provide a detailed explanation in Section 3.4 of the synthesis of a selection sort algorithm. The reader may find it useful to read the description and the example in parallel.

Let TYPEBAG(A) be the bag of primitive data types whose product forms the data type A. For example,

$$\text{TYPEBAG}(\mathbb{N} \times (\mathbb{N} \times \text{LIST}(\mathbb{N}))) = \{\mathbb{N}, \mathbb{N}, \text{LIST}(\mathbb{N})\}.$$

Assume we are given a specification $\overline{\Pi} = \langle D,R,I,O\rangle$. The following steps of the synthesis method produce a Σ-problem reduction representation $\hat{\overline{\Pi}} = \langle E,T,J,P\rangle$ representing $\overline{\Pi}$. The first step is to find a suitable sort set S and signature Σ.

1. Determine a set of sorts S and a simple S-sorted signature Σ of type $\langle w,\hat{s}\rangle$.

One heuristic for determining S and Σ is as follows. Let $a \in \text{TYPEBAG}(D)$ and $b \in \text{TYPEBAG}(R)$ be primitive types which are the principal carriers of known algebras A and B respectively where A and B have the same simple signature. Let S be the sort set and Σ be the S-sorted signature of A and B. Types a and b will be called the recurrent types in D and R respectively. Intuitively the recurrent types correspond to those inputs and outputs which will be decomposed and composed respectively. The other inputs and outputs will remain unaffected by the decomposition and composition operators.

For example, suppose $D = \mathbb{N} \times \text{LIST}(\mathbb{N})$ and $R = \text{BAGS}(\mathbb{N})$. According to Example 2.4.1 we have algebras with the same signature for LIST(ℕ) and BAG(ℕ) - in particular the sort set S is $\{\hat{s},c\}$ and the signature Σ has type $\langle c\hat{s},\hat{s}\rangle$. We adopt Σ as the signature of the algebras to be constructed on D and R.

2. Determine the component problems $\overline{\Pi}_s = \langle E_s,T_s,J_s,P_s\rangle$ for each $s \in S$.

Determining the principal problem $\overline{\Pi}_{\hat{s}}$ is easy since we want $\overline{\Pi}_{\hat{s}} = \overline{\Pi}$; i.e., $E_{\hat{s}} = D$, $T_{\hat{s}} = R$, $J_{\hat{s}} \Leftrightarrow I$, $P_{\hat{s}} \Leftrightarrow O$. The structure of the auxiliary problems is based on the simplifying assumption that a simple known function satisfies each auxiliary problem (alternatives are discussed in Section 3.5). Accordingly, for each $s \in S-\hat{s}$, let $E_s = A_s$ and $T_s = B_s$. We select a function f_s from the DSKB such

that $f_S:E_S \rightarrow T_S$, then let $J_S:x \Leftrightarrow \text{Defined}\cdot f_S:x$ and $P_S:\langle x,z\rangle \Leftrightarrow z = f_S:x$.

In the previous example we have sort set $S = \{\hat{S},c\}$, a signature with one operator of type $\langle c\hat{S},\hat{S}\rangle$, $E_{\hat{S}} = \mathbb{N} \times \text{LIST}(\mathbb{N})$, $T_{\hat{S}} = \text{Bag}(\mathbb{N})$ and algebras $A = \langle\{\mathbb{N},\text{LIST}(\mathbb{N})\},\{\text{Cons}\}\rangle$ and $B = \langle\{\mathbb{N},\text{BAG}(\mathbb{N})\},\{\text{Add}\}\rangle$. In these algebras $A_c = B_c = \mathbb{N}$, so we let $E_c = T_c = \mathbb{N}$. The single operator σ_E in E maps $E_{\hat{S}}$ into $E^{c\hat{S}}$ i.e.,

$$\sigma_E: \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times (\mathbb{N} \times \text{LIST}(\mathbb{N}))$$

and the single operator σ_T in T maps $T^{c\hat{S}}$ into $T_{\hat{S}}$, i.e.,

$$\sigma_T: \mathbb{N} \times \text{BAG}(\mathbb{N}) \rightarrow \text{BAG}(\mathbb{N}).$$

To determine the auxiliary problem $\hat{\amalg}_c$ we retrieve a known function mapping E_c to T_c ($\mathbb{N}$ to $\mathbb{N}$), such as the identity function Id. Then we can define $I_c:x \Leftrightarrow$ TRUE (i.e., Id is defined for all inputs) and $P_c:\langle x,z\rangle \Leftrightarrow \text{Id}:x = z$.

3. Construct a well-founded ordering on $E_{\hat{S}}$.

Choose a known well-founded ordering on $E_{\hat{S}}$. If none is known then construct a well-founded ordering $\succ$ on the recurrent type of $E_{\hat{S}}$ using Proposition 1, then if necessary use Proposition 2 to make it into a well-founded ordering on $E_{\hat{S}}$.

4. Construct the operators of E and T.

There are 2 alternate ways to finish the construction of E and T; in one we construct E then T, in the other we construct T then E. The important idea here is that once we construct the first operator we essentially "solve for" the output conditions of the other operator using the separability condition of Theorem 1. In other words, the separability condition can be likened to an equation in several unknowns - after plugging in a value for all variables but one we can solve for the value of the one. The "unknowns" are the output conditions of the operators σ_E, σ_T, and f_s for each $s \in S-\hat{S}$.

In the following sequence, called track ET, we construct E then T. The difference between the two tracks is this: the input and output types of the operators have already been determined. Furthermore, condition (1) of Theorem 1 predetermines some of the input and output conditions for σ_E. The only other

constraint on the operators is that they satisfy the separability condition. So in track ET we synthesize any function at all for σ_E satisfying the input/output type and condition (1) of Theorem 1 which should be easy to do. The hard part is determining an output condition for σ_T which satisfies the separability constraint. We use the precondition engine to find such an output condition then form a complete specification for σ_T. Finally, a function is synthesized satisfying this specification. In the other track, called track TE, an operator σ_T is synthesized satisfying a specification based on condition (2) of Theorem 1. we then derive an output condition for σ_E required by the separability constraint and use it to form a complete specification for σ_E. Finally σ_E is synthesized.

ET 4.1 Construct E

Construct a decomposition operator satisfying the specification

$$\sigma_E{:}x_0 = \langle x_1,\ldots,x_n\rangle \text{ such that } J_{\hat{s}}{:}x_0 \Rightarrow \bigwedge_{j \in \underline{n}} (J_{w_j}{:}x_j \wedge (w_j = \hat{s} \Rightarrow x_0 \rightarrowtail x_j))$$

$$\text{where } \sigma_E{:}E_{\hat{s}} \rightarrow E^w$$

with derived input condition $K_E{:}x_0$. The specification is constructed from condition (1) of Theorem 1 and the input/output types obtained from step 2. Included in the specification are all elements of the specification in condition (1) of Theorem 1 which are known at this point. After synthesizing the operator σ_E we can define $O_E{:}\langle x_0,x_1,\ldots,x_n\rangle$ to be $\sigma_E{:}x_0 = \langle x_1,\ldots,x_n\rangle$.

ET 4.2 Construct T.

ET 4.2.1 Derive the output conditions for σ_T.

Find a $\{z_0,z_1,\ldots,z_n\}$-precondition O_T of

$$\forall\langle z_0,z_1,\ldots,z_n\rangle \in T^{\hat{s}w}\ \forall\langle x_0,x_1,\ldots,x_n\rangle \in E^{\hat{s}w}$$
$$[\sigma_E{:}x_0 = \langle x_1,\ldots,x_n\rangle \wedge \bigwedge_{j \in \underline{n}} P_{w_j}{:}\langle x_j,z_j\rangle \Rightarrow P_{\hat{s}}{:}\langle x_0,z_0\rangle] \qquad \text{(ET 4.2.1)}$$

The derived precondition O_T is an output condition needed by σ_T in order to satisfy the separability condition of Theorem 1.

ET 4.2.2 Construct σ_T.

Using the precondition O_T from the previous step, we construct σ_T according to the specification

$$\sigma_T:\langle z_1,\ldots,z_n\rangle = z_0 \text{ such that } O_T:\langle z_0,z_1,\ldots,z_n\rangle$$
$$\text{where } \sigma_T:T^w \rightarrow T_{\hat{S}}.$$

This completes the description of track ET. We now present its alternate - track TE.

Track TE - Construct T then E

TE 4.1 Construct T

Construct the operator σ_T out of known functions according to the specification

$$\sigma_T:\langle z_1,\ldots,z_n\rangle = z_0 \text{ such that TRUE}$$
$$\text{where } \sigma_T:T^w \rightarrow T_{\hat{S}}.$$

All that we need to do here is to construct a mapping from T^w to $T_{\hat{S}}$. This specification is based on the specification in condition (2) of Theorem 1. Once σ_T has been synthesized we can define $O_T:\langle z_0,z_1,\ldots,z_n\rangle$ to be $\sigma_T:\langle z_1,\ldots,z_n\rangle = z_0$.

TE 4.2 Construct E.

TE 4.2.1 Derive output conditions for σ_E.

Find a $\{x_0,x_1,\ldots,x_n\}$-precondition O_E of

$$\forall\langle x_0,x_1,\ldots,x_n\rangle \in E^w \; \forall\langle z_0,z_1,\ldots,z_n\rangle \in T^{\hat{S}w}$$
$$[\underset{j\in\underline{n}}{\wedge} P_{w_j}:\langle x_j,z_j\rangle \wedge \sigma_T:\langle z_1,\ldots,z_n\rangle = z_0 \Rightarrow P_{\hat{S}}:\langle x_0,z_0\rangle] \qquad \text{(TE4.2.1)}$$

Taking O_E as an output condition of σ_E enables us to satisfy the separability condition of Theorem 1.

TE 4.2.2 Construct E

Construct the operator σ_E according to the specification

$$\sigma_E{:}x_0 = \langle x_1,\ldots,x_n\rangle \text{ such that } J_{\hat{s}}{:}x_0 \Rightarrow O_E{:}\langle x_0,x_1,\ldots,x_n\rangle \wedge$$
$$(\underset{j\in\underline{n}}{\wedge} J_{w_j}{:}x_j \wedge (w_j = \hat{s} \Rightarrow x_0 \rightarrow x_j))$$
$$\text{where } \sigma_E{:}E_{\hat{s}} \rightarrow E^w$$

with derived input condition $K_E{:}x$. This completes the description of step 4.

5. Determine the guard q.

In accordance with condition (5.2) of Theorem 1 the guard $\sim q$ is simply taken to be the derived input condition K_E returned by the construction of σ_E. We also attempt to verify condition (5.4) of Theorem 1 by deriving a $\{x\}$-precondition of

$$\forall x \in E_{\hat{s}}[J_{\hat{s}} \Rightarrow \text{Defined}\cdot q{:}x].$$

Let K_q be the derived precondition. If K_q is TRUE then condition (5.4) has been satisfied. Otherwise for legal inputs such that $J_{\hat{s}}{:}x \wedge K_q{:}x$ it is possible that the guard $q{:}x$ is undefined thus $f_{\hat{s}}{:}x$ is undefined. We take a simplified form of $J_{\hat{s}}{:}x \wedge K_q{:}x$ as a new input condition and return to step 4.

6. Construct the primitive operator.

Construct an operator h according to the specification

$$h{:}x_0 = z_0 \text{ such that } J_{\hat{s}}{:}x_0 \wedge q{:}x_0 \Rightarrow P_{\hat{s}}{:}\langle x_0,z_0\rangle$$
$$\text{where } h{:}E_{\hat{s}} \rightarrow T_{\hat{s}}.$$

with derived input condition K_h.

7. Construct a new input condition if necessary.

If h is unsatisfiable or if the derived input condition K_h is not TRUE then we are not guaranteed that h will handle correctly all inputs which it may be required to handle. It is necessary then to revise the input condition and then go back and rederive the operators of E and T, the guards, and h in accordance with the new input condition. At this point we have effectively derived a program satisfying output condition $P_{\hat{s}}$ with input condition

$$(J_{\hat{s}} \wedge \sim q) \vee (J_{\hat{s}} \wedge q \wedge K_h). \qquad (3.3.1)$$

We take a simplified form of (3.3.1) as the new input condition $J_{\hat{s}}$ and return to

step 4. In other words, formula (3.3.1) exactly describes the set of inputs which we know to have solutions thus we take it as our new input condition.

8. Assembly of a divide and conquer algorithm.

Assemble the functions derived above according to the schema

$$f_{\hat{S}}:x_0 \equiv \text{if } q:x_0 \rightarrow h:x_0 \;[]\; \neg q:x_0 \rightarrow \sigma_T \cdot f^W \cdot \sigma_E : x_0 \text{ fi.}$$

The derived input condition on this program is $J_{\hat{S}}:x_0$.

3.4 Synthesis of a Selection Sort Algorithm

3.4.1 Synthesis of Ssort

Suppose we are given the following specification for sorting a list of natural numbers

$$\text{Sort:}x = z \text{ such that } \text{Bag:}x = \text{Bag:}z \wedge \text{Ordered:}z$$
$$\text{where Sort:LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The input domains and output domains are both LIST(IN), the input condition is TRUE (i.e., there is none), and the output condition is Bag:x = Bag:z $\wedge$ Ordered:z. The steps of the synthesis method follow:

1. Determine a set of sorts S and an S-sorted signature $\sum$.

Since the input and output types are both LIST(IN) there is an easy choice of the recurrent type, namely LIST(IN). Several algebras are available in the DSKB with LIST(IN) as carrier; so suppose that we select A = <{IN,LIST(IN)}, {Cons}>. The sort set of A is S = {c,ŝ} where $A_{\hat{S}}$ = LIST(IN), A_c = IN, and the signature has type <cŝ,ŝ>, i.e., $\text{Cons:}A^{c\hat{S}} \rightarrow A_{\hat{S}}$.

2. Determine the component problems.

Let $E_{\hat{S}}$ = LIST(IN), $T_{\hat{S}}$ = LIST(IN), $J_{\hat{S}} \Leftrightarrow$ TRUE, $P_{\hat{S}}:\langle x,z\rangle \Leftrightarrow$ Bag:x = Bag:z $\wedge$ Ordered:z. In order to determine the auxiliary problem $\langle E_c,T_c,J_c,P_c\rangle$ we first

set $E_C = T_C = A_C = \mathbb{N}$. We then look for a simple function from E_C to T_C and select the identity function Id. Since Id is defined for all inputs we set

$$J_C{:}x_0 \iff \text{TRUE}$$
$$P_C{:}\langle x,z\rangle \iff z = Id{:}x \iff x = z$$

so that Id satisfies $\langle E_C, T_C, J_C, P_C\rangle$.

3. Construct a well-founded ordering on E_S.

Suppose that we do not have a known well-founded ordering on E_S. By Proposition 1 we can construct one based on a mapping from E_S to $\mathbb{N}$. The known function Length maps LIST($\mathbb{N}$) to $\mathbb{N}$ so define

$$x_0 \succ x_1 \text{ iff } Length{:}x_0 > Length{:}x_1.$$

By Proposition 1 $\langle E_S, \succ\rangle$ is a well-founded set.

4. Construct the operators of E and T.

Let us follow track TE and first construct T, then E.

TE 4.1 Construct T

The specification for σ_T is

$$\sigma_T{:}\langle b,z_1\rangle = z_0 \text{ such that TRUE}$$
$$\text{where } \sigma_T{:}\ \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The known function Cons has the same type as σ_T and we easily conclude then that

TE 4.2 Construct E

TE 4.2.1 Derive the output specification of σ_E

The output condition of σ_E must satisfy the separability condition so we set up the problem of finding a $\{x_0,a,x_1\}$-precondition of

$$\forall\langle x_0,a,x_1\rangle \in \text{LIST}(\mathbb{N}) \times \mathbb{N} \times \text{LIST}(\mathbb{N})\ \forall\langle z_0,b,z_1\rangle \in \text{LIST}(\mathbb{N}) \times \mathbb{N} \times \text{LIST}(\mathbb{N})$$
$$[\ a = b \wedge Bag{:}x_1 = Bag{:}z_1 \wedge Ordered{:}z_1 \wedge Cons{:}\langle b,z_1\rangle = z_0$$
$$\Rightarrow (Bag{:}x_0 = Bag{:}z_0 \wedge Ordered{:}z_0)]$$

To construct this formula we have made the following substitutions into the formula schema (TE 4.2.1):

1. replace w by $c\hat{s}$
2. replace $E_{\hat{s}}$ and $T_{\hat{s}}$ by LIST(ℕ)
3. replace $E^{c\hat{s}}$ and $T^{c\hat{s}}$ by $\mathbb{N} \times \text{LIST}(\mathbb{N})$
4. replace P_c:<a,b> by a = b
5. replace $P_{\hat{s}}$:<x,z> by Bag:x = Bag:z $\wedge$ Ordered:z
6. replace σ_T:<b,z_1> by Cons:<b,z_1>

In Figure 8 the precondition

$$a \leq \text{Bag}:x_1 \wedge \text{Bag}:x_0 = \text{Add}:\langle a, \text{Bag}:x_1 \rangle$$

is derived.

TE 4.2.2 Construct E

Using the output precondition derived above, σ_E is specified by

$$\sigma_E:x_0 = \langle a, x_1 \rangle \text{ such that } a \leq \text{Bag}:x_1 \wedge \text{Bag}:x_0 = \text{Add}:\langle a, \text{Bag}:x_0 \rangle \wedge \text{Length}:x_0 > \text{Length}:x_1$$
$$\text{where } \sigma_E:\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N})$$

In creating this specification we have simplified in certain ways: 1) since the input condition is TRUE it is omitted, 2) any conjunct which is TRUE is omitted, and 3) we replace $x_0 \succ x_1$ by its definition. In Section 3.4.2 we derive a program satisfying this specification with derived input condition $x_0 \neq \text{nil}$. For now we use the name Select in place of σ_E and assume that it can be synthesized with derived input condition $x_0 \neq \text{nil}$.

5. Determine the guards.

The guard $\neg q:x_0$ is simply the derived input condition $x_0 \neq \text{nil}$ required by Select. Consequently $q:x_0$ is $x_0 = \text{nil}$. We must also verify that the guard q is defined on all inputs satsifying the input condition. To do so we seek a $\{x_0\}$-precondition of

$$\forall x_0 \in \text{LIST}(\mathbb{N}) \; [\text{TRUE} \Rightarrow \text{Defined}:(x_0 = \text{nil})]$$

which is easily shown to be TRUE.

Hypotheses: h1. $a = b$

h2. $Bag{:}x_1 = Bag{:}z_1$

h3. $Ordered{:}z_1$

h4. $Cons{:}\langle b, z_1\rangle = z_0$

Variables: $\{x_0, a, x_1\}$

Goal 1:

$\langle Q1\rangle \quad Bag{:}x_0 = Bag{:}z_0$

R5 + h4

$\langle Q1\rangle \quad Bag{:}x_0 = Bag{:}Cons{:}\langle b, z_1\rangle$

R5 + L8

$\langle Q1\rangle \quad Bag{:}x_0 = Add{:}\langle b, Bag{:}z_1\rangle$

R5 + h2, R5 + h1

$\langle Q1\rangle \quad Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle$

P2

where Q1 is $Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle$

Goal 2:

$\langle Q2\rangle \quad Ordered{:}z_0$

R5 + h4

$\langle Q2\rangle \quad Ordered{:}Cons{:}\langle b, z_1\rangle$

R3 + L12

$\langle Q2\rangle \quad b \leq Bag{:}z_1$ $\qquad$ $\langle TRUE\rangle \quad Ordered{:}z_1$

R5 + h1 $\qquad$ P1 + h3

$\langle Q2\rangle \quad a \leq Bag{:}x_1$

P2

where Q2 is $a \leq Bag{:}x_1$

Figure 8: Derivation of the output condition of Select

6. Construct a primitive operator.

The specification for the primitive operator is

$$h{:}x_0 = z_0 \text{ such that } x_0 = \text{nil} \Rightarrow (\text{Bag}{:}x_0 = \text{Bag}{:}z_0 \wedge \text{Ordered}{:}z_0)$$
$$\text{where } h{:}\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N})$$

To create this specification from the specification schema for h in Section 3.3 we made the same substitutions as in step 4 plus the substitution of $x_0 = \text{nil}$ for q_0. When attempting to match known functions against a specification such as h the simplest functions are tried first. In this case the simplest of all, Id works. Proposition 4 can be used to verify that Id satisfies h.

7. Construct a new input condition.

In step 6 we found that Id satisfies the specification for h so no action is required here.

8. Assembly of divide and conquer algorithm.

Putting together all of the operators derived above, we obtain the following selection sort program:

$$\begin{aligned}\text{Ssort}{:}x_0 \equiv\ & \text{if} \\ & x_0 = \text{nil} \rightarrow x_0 \ [] \\ & x_0 \neq \text{nil} \rightarrow \text{Cons} \cdot (\text{Id} \times \text{Ssort}) \cdot \text{Select}{:}x_0 \\ & \text{fi}\end{aligned}$$

The derived input condition on Ssort is TRUE.

3.4.2 Synthesis of Select

In the previous section we derived the specification

$$\text{Select}{:}x_0 = \langle a, x_1 \rangle \text{ such that } \text{Bag}{:}x_0 = \text{Add}{:}\langle a, \text{Bag}{:}x_1 \rangle \wedge a \leq \text{Bag}{:}x_1 \wedge \text{Length}{:}x_0 > \text{Length}{:}x_1.$$
$$\text{where Select}{:}\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N})$$

Intuitively, $\text{Select}{:}x_0$ splits x_0 into two components, a and x_1, such that together a and x_1 have the same elements as x_0 and furthermore a is no greater than any element of x_1. Note that Select as specified has no input condition. However, for input nil the function is undefined. We will derive $x_0 \neq \text{nil}$ as an input condition of Select. The synthesis of Select proceeds as follows:

1. Determine a sort set S and signature Σ

The input type is LIST(IN) and the output type is IN $\times$ LIST(IN). We select LIST(IN) as recurrent type in both and again choose algebra $A = \langle \{IN, LIST(IN)\}, \{Cons\} \rangle$ in which LIST(IN) is the principal carrier. As above the sort set is $S = \{c, \hat{s}\}$, $A_c = IN$, $A_{\hat{s}} = LIST(IN)$ and the single operator symbol has the type $\langle c\hat{s}, \hat{s} \rangle$.

2. Determine the component problems.

Let $E_{\hat{s}} = LIST(IN)$, $T_{\hat{s}} = IN \times LIST(IN)$, $J_{\hat{s}}:x_0 \Leftrightarrow$ TRUE, and

$P_{\hat{s}}:\langle x_0, \langle a, x_1 \rangle \rangle \Leftrightarrow Bag:x_0 = Add:\langle a, Bag:x_1 \rangle \wedge a \leq Bag:x_1 \wedge Length:x_0 > Length\ x_1$.

Let $E_c = A_c = IN$ and $T_c = A_c = IN$. Next we select a known function f_s mapping E_c to T_c and again Id is the simpliest choice. Let $J_c:x_0$ be TRUE and $P_c:\langle x_0, x_1 \rangle$ be $x_0 = Id:x_1$ or simply $x_0 = x_1$.

3. Construct a well-founded ordering on $E_{\hat{s}}$.

$E_{\hat{s}} = LIST(IN)$ is made a well-founded set exactly as in the previous example by defining $x_0 \succ x_1$ iff $Length:x_0 > Length:x_1$.

ET 4. Construct the operators of E and T.

In this example let us construct E first then T.

ET 4.1 Construct σ_E

The decomposition function for Select must satisfy the incomplete specification

$$\sigma_E:x_0 = \langle u, x_1 \rangle \text{ such that } Length:x_0 > Length:x_1$$
$$\text{where } \sigma_E:LIST(IN) \rightarrow IN \times LIST(IN).$$

This specification has been constructed according to the specification schema in step ET 4.1 of the synthesis method. We show only the simplified result. In constructing this specification we have omitted the input condition (since it is TRUE) and various output conditions which are TRUE. This practice will be followed in the sequel. A function structure is needed here and Proposition 4 can be used to show that [First,Rest] satisfies σ_E with derived input condition $x_0 \neq nil$. First, the input and output domains of [First,Rest] are identical to that for σ_E. By Axiom L6 we have $x \neq nil$ as the domain of definition of

[First,Rest]. In terms of Proposition 4 then we have:

$I_1{:}x_0 \Leftrightarrow x_0 \neq \text{nil}$

$I_2{:}x_0 \Leftrightarrow \text{TRUE}$

$O_2{:}\langle x_0, \langle u, x_1\rangle\rangle \Leftrightarrow \text{Length}{:}x_0 > \text{Length}{:}x_1$.

According to condition (d) of Proposition 4 we derive a $\{x_0\}$-precondition of

$$\forall x \in \text{LIST}(\mathbb{N})\ [\text{TRUE} \Rightarrow x_0 \neq \text{nil}]$$

which is simply $x_0 \neq \text{nil}$ (called J in Proposition 4). Finally, according to condition (e) of Proposition 4, we derive TRUE as an $\{x_0\}$-precondition of

$\forall x_0 \in \text{LIST}(\mathbb{N})\ \forall \langle u, x_1\rangle \in \mathbb{N} \times \text{LIST}(\mathbb{N})$
$[x_0 \neq \text{nil} \wedge \text{TRUE} \wedge [\text{First,Rest}]{:}x_0 = \langle u, x_1\rangle \Rightarrow \text{Length}{:}x_0 > \text{Length}{:}x_1]$.

in Figure 9. Thus by Proposition 4 [First,Rest] satisfies σ_E with derived input condition $x_0 \neq \text{nil} \wedge \text{TRUE}$, or simply $x_0 \neq \text{nil}$.

ET 4.2 Construct T.

Hypotheses: h1. $x_0 \neq \text{nil}$
h2. $[\text{First,Rest}]{:}x_0 = \langle u, x_1\rangle$
h3. $u = \text{First}{:}x_0$
h4. $x_1 = \text{Rest}{:}x_0$

Variables: $\{x_0\}$

Goal 1:

$\langle \text{TRUE}\rangle\ \text{Length}{:}x_0 > \text{Length}{:}x_1$
| R6 + h2 + L7
$\langle \text{TRUE}\rangle\ \text{Length}\cdot\text{Cons}{:}\langle u, x_1\rangle > \text{Length}{:}x_1$
| R5 + L15
$\langle \text{TRUE}\rangle\ 1 + \text{Length}{:}x_1 > \text{Length}{:}x_1$
| R3 + n1
$\langle \text{TRUE}\rangle\ 1 > 0$
P1

Figure 9: Matching [First,Rest] with specification for σ_E.

ET 4.2.1 Derive the output condition of σ_T.

The output condition of σ_T must satisfy the separability condition so we seek a $\{a_0,z_0,v,a_1,z_1\}$-precondition of

$$\forall <<a_0,z_0>,v,<a_1,z_1>> \in \mathbb{N} \times LIST(\mathbb{N})) \times \mathbb{N} \times (\mathbb{N} \times LIST(\mathbb{N}))$$
$$\forall <x_0,u,x_1> \in LIST(\mathbb{N}) \times \mathbb{N} \times LIST(\mathbb{N})$$
$$[[First,Rest]:x_0 = <u,x_1> \wedge u = v \wedge Bag:x_1 = Add:<a_1,Bag:z_1> \wedge a_1 \leq Bag:z_1 \wedge$$
$$Length:x_1 > Length:z_1 \Rightarrow$$
$$(Bag:x_0 = Add:<a_0,z_0> \wedge a_0 \leq Bag:z_0> \wedge Length:x_0 > Length:z_0)].$$

To create this formula the following substitutions were made on the formula schema in step ET 4.2.1 of the synthesis method:

$c\hat{s}$ replaces w

$E_{\hat{s}} = LIST(\mathbb{N})$ and $T_{\hat{s}} = \mathbb{N} \times LIST(\mathbb{N})$

$E_c = T_c = \mathbb{N}$

[First,Rest] replaces σ_E

Id replaces P_c

$Bag:x_1 = Add:<a_1,Bag:z_1> \wedge a_1 \leq Bag:z_1 \wedge Length:x_1 > Length:z_1$
replaces $P_{\hat{s}}:<x_i,z_i>$

In Figure 10, the precondition

$$a_1 \leq Bag:z_1 \Rightarrow Add:<v, Add:<a_1,Bag:z_1>> = Add:<a_0,Bag:z_0> \wedge$$
$$a_0 \leq Bag:z_0 \wedge 2 + Length:z_1 > Length:z_0$$

is derived.

ET 4.2.2 Construct σ_T.

We construct the specification

$$\sigma_T:<v,<a_1,z_1>> = <a_0,z_0> \text{ such that } a_1 \leq Bag:z_1 \Rightarrow a_0 \leq Bag:z_0 \wedge$$
$$Add:<v, Add:<a_1,Bag:z_1>> = Add:<a_0,Bag:z_0> \wedge 2 + Length:z_1 > Length:z_0$$
$$\text{where } \sigma_T: \mathbb{N} \times (\mathbb{N} \times LIST(\mathbb{N})) \rightarrow \mathbb{N} \times LIST(\mathbb{N})$$

A conditional program, call it Compose, can be constructed satisfying this specification with derived input condition TRUE:

Hypotheses: h1. $[First,Rest]:x_0 = \langle u,x_1 \rangle$

h2. $Bag:x_1 = Add:\langle a_1,Bag:z_1 \rangle$

h3. $b_1 \leq Bag:z_1$

h4. $u = v$

h5. $Length:x_1 > Length:z_1$

Variables: $\{a_1,z_1,v,a_0,z_0\}$

Goal 1:

$\langle Q1 \rangle \quad Bag:x_0 = Add:\langle a_0,Bag:z_0 \rangle$

R6 + h1 + L7

$\langle Q1 \rangle \quad Bag:Cons:\langle u,x_1 \rangle = Add:\langle a_0,Bag:z_0 \rangle$

R5 + L8

$\langle Q1 \rangle \quad Add:\langle u,Bag:x_1 \rangle = Add:\langle a_0,Bag:z_0 \rangle$

R5 + h4, R5 + h2

$\langle Q1 \rangle \quad Add:\langle v,Add:\langle a_1,Bag:z_1 \rangle \rangle = Add:\langle a_0,Bag:z_0 \rangle$

P2

where Q1 is $Add:\langle v,Add:\langle a_1,Bag:z_1 \rangle \rangle = Add:\langle a_0,Bag:z_0 \rangle$

Goal 2:

$\langle a_0 \leq Bag:z_0 \rangle \quad a_0 \leq Bag:z_0$

P2

Figure 10a: Derivation of output conditions for σ_T.

Goal 3:

$$
\begin{array}{c}
\langle Q2\rangle \quad Length:x_0 > Length:z_0 \\
\big|\ R6 + h1 + L7 \\
\langle Q2\rangle \quad Length:Cons:\langle u,x_1\rangle > Length:z_0 \\
\big|\ R5 + L15 \\
\langle Q2\rangle \quad 1 + Length:x_1 > Length:z_0 \\
\big|\ R6 + h2 + L19 \\
\langle Q2\rangle \quad 1 + Card:Add:\langle a_1,Bag:z_1\rangle > Length:z_0 \\
\big|\ R5 + B6 \\
\langle Q2\rangle \quad 1 + 1 + Card:Bag:z_1 > Length:z_0 \\
\big|\ R5 + L18 \\
\langle Q2\rangle \quad 1 + 1 + Length:z_1 > Length:z_0 \\
P2
\end{array}
$$

where Q2 is $a_1 \leq Bag:z_1 \Rightarrow 1 + 1 + Length:z_1 > Length:z_0$

Figure 10b: Derivation of output conditions for σ_T.

$$
\begin{array}{l}
Compose:\langle v,\langle a_1,z_1\rangle\rangle \equiv \text{if} \\
\qquad v \leq a_1 \rightarrow \langle v,Cons:\langle a_1,z_1\rangle\rangle \,[\!] \\
\qquad v \geq a_1 \rightarrow \langle a_1,Cons:\langle v,z_1\rangle\rangle \\
\text{fi}
\end{array}
$$

5. Determine the guards.

The input condition derived for σ_E is $x_0 \neq nil$ which we take as $\sim q:x_0$. To verify condition (5.4) of Theorem 1 we easily prove that ~q is defined on legal inputs. However, we noted before that Select will be undefined when its input is nil, yet here we are about to use the guard $x_0 = nil$ on the primitive operator. The next step will reveal the need for revision of the input condition.

6. Construct a primitive operator.

The specification of the primitive operator is

$$h{:}x_0 = \langle a, x_1\rangle \text{ such that } x_0 = nil \Rightarrow Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle \wedge a \leq Bag{:}x_1 \wedge Length{:}x_0 > Length{:}x_1$$
$$\text{where } h{:}LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N}).$$

It is easily shown that this specification is unsatisfiable.

7. Construction of a new input condition.

Since the specification for the primitive operator is unsatisfiable we form a new input condition by constructing and simplifying the expression

$$(TRUE \wedge \sim(x_0 = nil)) \vee (TRUE \wedge x_0 = nil \wedge FALSE)$$

yielding $x_0 \neq nil$. In effect we exclude nil as a legal input to Select and return to an earlier stage in the synthesis and rederive σ_E, σ_T, and q. In the following we retrace some of the previous steps:

4'. Construct E then T.

The input condition $J_S{:}x_0$ is redefined to be $x_0 \neq nil$.

ET 4.1' Construct σ_E

The new specification for σ_E is

$$\sigma_E{:}x_0 = \langle u, x_1\rangle \text{ such that } x_0 \neq nil \Rightarrow Length{:}x_0 > Length{:}x_1 \wedge x_1 \neq nil$$
$$\text{where } \sigma_E{:}LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N}).$$

We found that [First,Rest] satisfied the earlier specification for σ_E so it is reasonable to try it again. This time we have (in terms of Proposition 4)

$I_1{:}x_0 \Leftrightarrow x_0 \neq nil$

$I_2{:}x_0 \Leftrightarrow x_0 \neq nil$

$O_2{:}\langle x_0, \langle a, x_1\rangle\rangle \Leftrightarrow (Length{:}x_0 > Length{:}x_1 \wedge x_1 \neq nil)$

In satisfying condition (d) of Proposition 4 we derive TRUE as an $\{x_0\}$-precondition of

$$\forall x_0 \in LIST(\mathbb{N})\ [x_0 \neq nil \Rightarrow x_0 \neq nil].$$

In satisfying condition (e) we set up the problem of finding an $\{x_0\}$-precondition of

$$\forall x_0 \in LIST(\mathbb{N})\ \forall \langle a, x_1\rangle \in \mathbb{N} \times LIST(\mathbb{N})\ [x_0 \neq nil \wedge a = First{:}x_0 \wedge x_1 = Rest{:}x_0 \Rightarrow Length{:}x_0 > Length{:}x_1 \wedge x_1 \neq nil].$$

The precondition $\text{Rest}:x_0 \neq \text{nil}$ is derived in Figure 11. By Proposition 4 [First,Rest] satisfies σ_E with derived input condition $\text{Rest}:x_0 \neq \text{nil}$.

ET 4.2'. Construct T.

This step does not involve the input condition so the derivation of σ_T proceeds as before - we synthesize the conditional program Compose.

5'. Determine the guard.

The derived input condition on σ_E is $\text{Rest}:x_0 \neq \text{nil}$ thus $q:x_0$ is $\text{Rest}:x_0 = \text{nil}$. In Figure 12 we verify that q is defined when the input condition $(x_0 \neq \text{nil})$ holds by deriving TRUE as a $\{x_0\}$-precondition of

$$\forall x_0 \in \text{LIST}(\mathbb{N})\ [x_0 \neq \text{nil} \Rightarrow \text{Defined}:(\text{Rest}:x_0 = \text{nil})].$$

6'. Construction of a primitive operator.

The new specification for the primitive operator is

Hypotheses: h1. $x_0 \neq \text{nil}$
h2. $\text{First}:x_0 = a$
h3. $\text{Rest}:x_0 = x_1$

Variables: $\{x_0\}$

Goal 1: $\langle \text{TRUE} \rangle\ \text{Length}:x_0 > \text{Length}:x_1$
(see derivation in Figure 9)

Goal 2:

$$\begin{array}{l} \langle \text{Rest}:x_0 \neq \text{nil} \rangle\ x_1 \neq \text{nil} \\ \qquad\qquad\qquad\qquad \Big| \ \text{R5} + \text{h3} \\ \langle \text{Rest}:x_0 \neq \text{nil} \rangle\ \text{Rest}:x_0 \neq \text{nil} \\ \qquad\qquad\qquad\qquad \text{P2} \end{array}$$

Figure 11: Matching [First,Rest] with specifications for σ_E.

Hypotheses: h1. $x_0 \neq nil$

Variables: {}

Goal 1: <TRUE> Defined:(Rest:x_0 = nil)

$\wedge$ R3 + L3, R1

<TRUE> Defined:Rest:x_0 <TRUE> Defined:nil

R3 + L5 P1 + L1

<TRUE> $x_0 \neq nil$

P1 + h1

Figure 12: Verifying the quard in Select.

$$h{:}x_0 = \langle a, x_1\rangle \text{ such that } Rest{:}x_0 = nil \Rightarrow Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle \wedge a \leq Bag{:}x_1 \wedge Length{:}x_0 > Length{:}x_1$$

where $h{:}LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N})$.

The function structure [First,Rest] is easily found to satisfy this specification as follows: Again in terms of Proposition 4 we have:

$$?_2{:}x_0 \Leftrightarrow Rest{:}x_0 = nil$$

$$?_2{:}\langle x_0, \langle a, x_1\rangle\rangle \Leftrightarrow Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle \wedge a \leq Bag{:}x_1 \wedge Length{:}x_0 > Length{:}x_1$$

$$?_1{:}x_0 \Leftrightarrow x_0 \neq nil \text{ (by Axiom L6)}$$

To satisfy condition (d) of Theorem 1, TRUE is derived as an $\{x_0\}$-precondition of

$$\forall x \in LIST(\mathbb{N})[Rest{:}x_0 = nil \Rightarrow x_0 \neq nil]$$

in Figure 13. Finally, to satisfy condition (e), we set up the problem of finding a $\{x_0\}$-precondition of

$$?\langle x_0, \langle a_1, x_1\rangle\rangle \in LIST(\mathbb{N}) \times (\mathbb{N} \times LIST(\mathbb{N}))$$

$$[TRUE \wedge Rest{:}x_0 = nil \wedge First{:}x_0 = a_1 \wedge Rest{:}x_0 = x_1 \Rightarrow Bag{:}x_0 = Add{:}\langle a, Bag{:}x_1\rangle \wedge a \leq Bag{:}x_1 \wedge Length{:}x_0 > Length{:}x_1].$$

The precondition TRUE is derived in Figure 14. Finally by Proposition 4 we

Hypotheses: h1. Rest:x_0 = nil

Variables: {}

Goal 1:

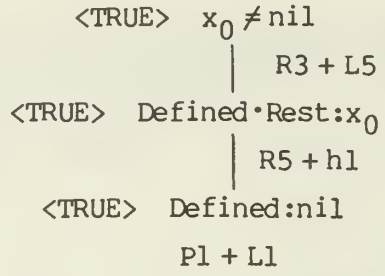


Figure 13: Matching [First,Rest] with the specification for h.

Hypotheses: h1. $\text{Rest}:x_0 = \text{nil}$

h2. $[\text{First},\text{Rest}]:x_0 = \langle a, x_1 \rangle$

h3. $\text{First}:x_0 = a$

h4. $\text{Rest}:x_0 = x_1$

Variables: $\{x_0\}$

Goal 1:

$$\begin{array}{c} \langle \text{TRUE} \rangle \quad \text{Bag}:x_0 = \text{Add}:\langle a, \text{Bag}:x_1 \rangle \\ \Big| \; \text{R6} + \text{h2} + \text{L7} \\ \langle \text{TRUE} \rangle \quad \text{Bag}:\text{Cons}:\langle a, x_1 \rangle = \text{Add}:\langle a, \text{Bag}:x_1 \rangle \\ \Big| \; \text{R5} + \text{L8} \\ \langle \text{TRUE} \rangle \quad \text{Add}:\langle a, \text{Bag}:x_1 \rangle = \text{Add}:\langle a, \text{Bag}:x_1 \rangle \\ \text{P1} + \text{B1} \end{array}$$

2.

$$\begin{array}{c} \langle \text{TRUE} \rangle \quad a \leq \text{Bag}:x_1 \\ \Big| \; \text{R5} + \text{h4} \\ \langle \text{TRUE} \rangle \quad a \leq \text{Bag}:\text{Rest}:x_0 \\ \Big| \; \text{R5} + \text{h1} \\ \langle \text{TRUE} \rangle \quad a \leq \text{Bag}:\text{nil} \\ \Big| \; \text{R5} + \text{L20} \\ \langle \text{TRUE} \rangle \; a \leq \emptyset \\ \text{P1} + \text{B2} \end{array}$$

Goal 3:

$$\begin{array}{c} \langle \text{TRUE} \rangle \quad \text{Length}:x_0 > \text{Length}:x_1 \\ \Big| \; \text{R6} + \text{h2} + \text{L7} \\ \langle \text{TRUE} \rangle \quad \text{Length}:\text{Cons}:\langle a, x_1 \rangle > \text{Length}:x_1 \\ \Big| \; \text{R5} + \text{L8} \\ \langle \text{TRUE} \rangle \quad 1 + \text{Length}:x_1 > \text{Length}:x_1 \\ \Big| \; \text{R3} + \text{n1} \\ \langle \text{TRUE} \rangle \quad 1 > 0 \\ \text{P1} \end{array}$$

Figure 14: Verifying the match of [First,Rest] with h.

conclude that [First, Rest] satisfies h with derived input condition $\text{Rest}:x_0 = \text{nil} \wedge \text{TRUE} \wedge \text{TRUE}$ or simply $\text{Rest}:x_0 = \text{nil}$.

7'. Derivation of a new input condition.

Since we constructed an operator satisfying specification h this step is bypassed.

8'. Construction of a divide and conquer algorithm.

The functions derived above are assembled into the following program:

$$\begin{aligned}
&\text{Select}:x_0 \equiv \text{if}\\
&\qquad \text{Rest}:x_0 = \text{nil} \rightarrow [\text{First},\text{Rest}]:x_0 \ \square\\
&\qquad \text{Rest}:x_0 \neq \text{nil} \rightarrow \text{Compose}\cdot(\text{Id} \times \text{Select})\cdot[\text{First},\text{Rest}]:x_0\\
&\quad \text{fi}
\end{aligned}$$

The derived input condition on Select is $x_0 \neq \text{nil}$ $(J_{\hat{S}})$. The complete selection sort program synthesized in the above examples is listed in Figure 15.

$$\begin{aligned}
&\text{Ssort}:x_0 \equiv \text{if}\\
&\qquad x_0 = \text{nil} \rightarrow x_0 \ \square\\
&\qquad x_0 \neq \text{nil} \rightarrow \text{Cons}\cdot(\text{Id} \times \text{Ssort})\cdot\text{Select}:x_0\\
&\quad \text{fi}
\end{aligned}$$

$$\begin{aligned}
&\text{Select}:x \equiv \text{if}\\
&\qquad \text{Rest}:x = \text{nil} \rightarrow [\text{First},\text{Rest}]:x \ \square\\
&\qquad \text{Rest}:x \neq \text{nil} \rightarrow \text{Compose}\cdot(\text{Id} \times \text{Select})\cdot[\text{First},\text{Rest}]:x\\
&\quad \text{fi}
\end{aligned}$$

$$\begin{aligned}
&\text{Compose}:\langle v_1,\langle v_2,z\rangle\rangle \equiv \text{if}\\
&\qquad v_1 \leq v_2 \rightarrow \langle v_1,\text{Cons}:\langle v_2,z\rangle\rangle \ \square\\
&\qquad v_1 \geq v_2 \rightarrow \langle v_2,\text{Cons}:\langle v_1,z\rangle\rangle\\
&\quad \text{fi}
\end{aligned}$$

Figure 15: A Selection Sort Program

3.5 Remarks on the Synthesis Method

For the sake of simplifying the presentation we have placed a number of restrictions on the synthesis method. First, we only consider algebras E and T with a single operator, thus the term "simple divide and conquer algorithms". It is not hard to relax this constraint but the analogue of Theorem 4 becomes more complex. Second, we assume that the auxiliary problems are simply solved by primitive functions in the target language. A design method can be devised which allows the synthesis of nonprimitive auxiliary functions in the following way. The essence of the design method in Section 3.3 is the use of the separability condition to solve for the output conditions of either σ_E or σ_T. We use it to set up a precondition problem by assuming simple solutions for the auxiliary problems and for, say, σ_E, then we derive output conditions for σ_T. The separability condition of Theorem 1 can be likened to a linear equation in several variables. We plug in simple values for all but one variable x then solve for x. We could just as well plug in simple decomposition and composition operators and solve for the output conditions of the auxiliary problems. Third, we assume that there is only a single recurrent type: i.e., that only one of the parameters to a divide and conquer algorithm will be decomposed. It is not hard to relax this restriction however the decision about which parameters to decompose is not well-motivated at present (relying on rather weak heuristics).

We now relate our synthesis method with previous work on deductive program synthesis. Suppose the user supplies a specification $\overline{\Pi} = \langle D,R,I,O\rangle$. The deductive approach to program synthesis [16,17,4] seeks to extract a program f from a constructive proof of the theorem

$$\forall x \in D \; \exists z \in R \; [I{:}x \Rightarrow O{:}\langle x,z\rangle] \qquad (3.5.1)$$

Theorem proving techniques, more or less adapted to the special demands of program synthesis, are used to prove the theorem constructively. We advance this approach in two ways. First, our definition of the program synthesis problem is slightly more general. We seek to extract a program f from a constructive derivation of an {x}-precondition of (3.5.1). The resulting precondition is the derived input condition. With this approach it is easier to create specifications because we are no longer required to completely specify the input condition. It also facilitates top-down problem decomposition because the decomposition process is not obliged to create complete specifications for subproblems. Second, rather than rely on general theorem proving techniques we emphasize the use of special purpose program synthesis-oriented techniques,

specifically the design methods. The design method for simple divide and conquer algorithms is based on Theorem 1 but it also embodies the procedural knowledge needed to apply it. Viewed as a theorem proving tool it seeks to exploit special structure in the specification. Specifically, it seeks to decompose the problem structure with respect to the separability condition of Theorem 1. We are currently developing analogous design methods for other classes of algorithms.

4. Further Examples

In this section we derive three other sorting algorithms whose structure is that of divide and conquer. These algorithms are all constructed from the same specification but their design diverges when distinct choices are made at certain steps in the synthesis method.

4.1 Synthesis of an Insertion Sort

4.1.1 Synthesis of Isort

The synthesis of an insertion sort and a selection sort are similar. The synthesis process diverges at Step 4 - the selection sort follows track TE, insertion sort follows track ET. Intuitively, selection sorts make use of a simple composition operator and a somewhat complex decomposition operator whereas insertion sorts make use of a simple decomposition operator and a complex composition operator. Again the initial specification for sorting a list of natural numbers is

$$\text{Sort:}x = z \text{ such that } \text{Bag:}x = \text{Bag:}z \wedge \text{Ordered:}z$$
$$\text{where Sort:LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

Isort is synthesized as follows with some details omitted for brevity.

1,2,3. Determine a sort set, signature, component problems, and a well-founded ordering.

As in Ssort let $S = \{c, \hat{S}\}$, let $\sum$ be a simple S-sorted signature of type $\langle c\hat{S}, \hat{S}\rangle$. and let $E_{\hat{S}} = \text{LIST}(\mathbb{N})$, $T_{\hat{S}} = \text{LIST}(\mathbb{N})$, $J_{\hat{S}}:x \Leftrightarrow \text{TRUE}$, and $P_{\hat{S}}:\langle x,z\rangle \Leftrightarrow$ Bag:x=Bag:z $\wedge$ Ordered z.

Also let $E_c = \mathbb{N}$, $T_c = \mathbb{N}$, $f_c:x = x$, and thus again $J_c:x \Leftrightarrow \text{TRUE}$, $P_c:\langle x,z\rangle \Leftrightarrow x = z$.

Define $x_0 \succ x_1$ by Length:x_0>Length:x_1.

At step 4 the synthesis process follows track ET.

ET 4.1 Construct E.

The decomposition operator σ_E for Isort has the specification

$$\sigma_E{:}x_0 = \langle a, x_1\rangle \text{ such that } \text{Length}{:}x_0 > \text{Length}{:}x_1$$
$$\text{where } \sigma_E{:}\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N}).$$

In constructing this specification we have omitted the input condition since it is TRUE and various output conditions which are TRUE. This practice will be followed in the sequel. The construction [First,Rest] is easily shown to satisfy this specification with derived input condition $x_0 \neq \text{nil}$. (See the analogous matching process in Figure 9).

ET 4.2 Construct T.

ET 4.2.1 Derive output conditions of σ_T.

The output condition of σ_T is obtained by deriving a $\{z_0, b, z_1\}$-precondition of

$$\forall \langle z_0, \langle b, z_1\rangle\rangle \in \text{LIST}(\mathbb{N}) \times (\mathbb{N} \times \text{LIST}(\mathbb{N}))\ \forall \langle x_0, \langle a, x_1\rangle\rangle \in \text{LIST}(\mathbb{N}) \times (\mathbb{N} \times \text{LIST}(\mathbb{N}))$$
$$|[\text{First,Rest}]{:}x_0 = \langle a, x_1\rangle \wedge a = b \wedge \text{Bag}{:}x_1 = \text{Bag}{:}z_1 \wedge \text{Ordered}{:}z_1$$
$$\Rightarrow \text{Bag}{:}x_0 = \text{Bag}{:}z_0 \wedge \text{Ordered}{:}z_0].$$

The precondition

$$\text{Ordered}{:}z_1 \Rightarrow \text{Add}{:}\langle b, \text{Bag}{:}z_1\rangle = \text{Bag}{:}z_0 \wedge \text{Ordered}{:}z_0$$

is derived in Figure 16.

ET 4.2.2 Construct T

Using the precondition derived in the previous step we create the specification

$$\sigma_T{:}\langle b, z_1\rangle = z_0 \text{ such that } \text{Ordered}{:}z_1 \Rightarrow \text{Add}{:}\langle b, \text{Bag}{:}z_1\rangle = \text{Bag}{:}z_0 \wedge \text{Ordered}{:}z_0$$
$$\text{where } \sigma_T{:}\ \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

A program called Insert is synthesized in the following section which satisfies this specification.

5. Determine the guard.

The guard $\sim q{:}x_0$ is simply the derived input condition $x_0 \neq \text{nil}$ on σ_E. To verify that q is defined on legal inputs we easily can show that

$$\forall x \in \text{LIST}(\mathbb{N})[\text{TRUE} \Rightarrow \text{Defined}{:}(x = \text{nil})]$$

Hypotheses: h1. $[First,Rest]:x_0 = \langle a,x_1 \rangle$

h2. $a = b$

h3. $Bag:x_1 = Bag:z_1$

h4. $Ordered:z_1$

Variables: $\{b,z_0,z_1\}$

Goal 1:

$$\langle Q \rangle \quad Bag:x_0 = Bag:z_0$$
$$\mid \quad R6 + h1 + L7$$
$$\langle Q \rangle \quad Bag:Cons:\langle a,x_1 \rangle = Bag:z_0$$
$$\mid \quad R5 + L8$$
$$\langle Q \rangle \quad Add:\langle a,Bag:x \rangle = Bag:z_0$$
$$\mid \quad R5 + h2$$
$$\langle Q \rangle \quad Add:\langle b,Bag:z_1 \rangle = Bag:z_0$$
$$P2$$

where Q is $Ordered:z_1 \Rightarrow Add:\langle b,Bag:z_1 \rangle = Bag:z_0$

Goal 2:

$$\langle Ordered:z_1 \Rightarrow Ordered:z_0 \rangle \; Ordered:z_0$$
$$P2$$

Figure 16: Derivation of Output Conditions for the Composition Operator of Isort

is valid.

6. Construct the primitive operator.

The primitive operator has specification

$$h:x_0 = z_0 \text{ such that } x_0 = nil \Rightarrow (Bag:x_0 = Bag:z_0 \wedge Ordered:z_0).$$
$$\text{where } h:LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}).$$

As in the synthesis of Ssort we can show that Id satisfies h.

7. Construct new input condition.

Since we constructed an operator satisfying the specification for h this step is bypassed.

8. Construction of the divide and conquer algorithm.

Putting together the operators derived above we obtain

$$\begin{aligned} \text{Isort:}x \equiv\ & \text{if} \\ & x = \text{nil} \rightarrow x\ [\,] \\ & x \neq \text{nil} \rightarrow \text{Insert}\cdot(\text{Id} \times \text{Isort})\cdot[\text{First},\text{Rest}]:x \\ & \text{fi}. \end{aligned}$$

The derived input condition on Isort is simply TRUE.

4.1.2 Synthesis of Insert

The following specification for Insert was derived in the preceeding section

$$\text{Insert}:\langle a_0,x_0\rangle = z_0 \text{ such that } \text{Ordered}:x_0 \Rightarrow \text{Bag}:z_0 = \text{Add}:\langle a_0,\text{Bag}:x_0\rangle \wedge \text{Ordered}:z_0$$
$$\text{where Insert}: \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The task of Insert is to place a number in an ordered list such that the resulting list is ordered. The synthesis of Insert proceeds as follows.

1,2 Determine a sort set, signature, and component problems.

Choices like those made in Ssort, Select, and Isort can be made here for Insert with the result $S = \{c,\hat{s}\}$, $\sum$ is a simple S-sorted signature of type $\langle c\hat{s},\hat{s}\rangle$,

$$E_{\hat{s}} = \mathbb{N} \times \text{LIST}(\mathbb{N})$$
$$T_{\hat{s}} = \text{LIST}(\mathbb{N})$$
$$J_{\hat{s}}:\langle a_0,x_0\rangle \Leftrightarrow \text{Ordered}:x_0$$
$$P_{\hat{s}}:\langle\langle a_0,x_0\rangle,z_0\rangle \Leftrightarrow \text{Bag}:z_0 = \text{Add}:\langle a_0,\text{Bag}:x_0\rangle \wedge \text{Ordered}:x_0,$$
$$E_c = \mathbb{N}, \text{ and } T_c = \mathbb{N}.$$

Again we find $\text{Id}:E_c \rightarrow T_c$ so let

$$J_c \Leftrightarrow \text{TRUE},$$
$$P_c:\langle a_1,b\rangle \Leftrightarrow a_1 = b, \text{ and}$$
$$f_c \equiv \text{Id}.$$

3. Determine a well-founded ordering on $E_{\hat{s}}$.

Using Propositions 1 and 2, we can construct a well-founded ordering on the input type $\mathbb{N} \times \text{LIST}(\mathbb{N})$ defined by

$$\langle a_0, x_0 \rangle \succ \langle a_1, x_1 \rangle \text{ iff } \text{Length}:x_0 > \text{Length}:x_1.$$

TE 4. Construct T then E

TE 4.1 Construct T.

The composition operator of Insert has partial specification

$$\sigma_T:\langle b, z_1 \rangle = z_0 \text{ such that TRUE}$$
$$\text{where } \sigma_T: \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N})$$

The primitive operator Cons is easily shown to satisfy this specification.

TE 4.2 Construct E.

TE 4.2.1 Derive output conditions for σ_E.

The output conditions of σ_E are found by deriving a $\{a_0, x_0, a_1, a_2, x_2\}$-precondition of

$$\forall \langle\langle a_0, x_0 \rangle, a_1, \langle a_2, x_2 \rangle\rangle \in (\mathbb{N} \times \text{LIST}(\mathbb{N})) \times \mathbb{N} \times (\mathbb{N} \times \text{LIST}(\mathbb{N}))$$
$$\forall \langle z_0, b, z_1 \rangle \in \text{LIST}(\mathbb{N}) \times \mathbb{N} \times \text{LIST}(\mathbb{N})$$
$$[a_1 = b \wedge \text{Bag}:z_1 = \text{Add}:\langle a_2, \text{Bag}:x_2 \rangle \wedge \text{Ordered}:z_1 \wedge \text{Cons}:\langle b, z_1 \rangle = z_0$$
$$\Rightarrow \text{Bag}:z_0 = \text{Add}:\langle a_0, \text{Bag}:x_0 \rangle \wedge \text{Ordered}:z_0].$$

The derivation in Figure 17 yields precondition

$$\text{Add}:\langle a_1, \text{Add}:\langle a_2, \text{Bag}:x_2 \rangle\rangle = \text{Add}:\langle a_0, \text{Bag}:x_0 \rangle \wedge a_1 \leq a_2 \wedge a_1 \leq \text{Bag}:x_2.$$

TE 4.2.2 Construct E.

We construct specification

$$\sigma_E:\langle a_0, x_0 \rangle = \langle a_1, \langle a_2, x_2 \rangle\rangle \text{ such that } \text{Length}:x_0 > \text{Length}:x_2 \wedge$$
$$\text{Add}:\langle a_1, \text{Add}:\langle a_2, \text{Bag}:x_2 \rangle\rangle = \text{Add}:\langle a_0, \text{Bag}:x_0 \rangle \wedge a_1 \leq a_2 \wedge a_1 \leq \text{Bag}:x_2$$
$$\text{where } \sigma_E: \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times (\mathbb{N} \times \text{LIST}(\mathbb{N})).$$

A simple conditional program called Decompose is easily constructed according to this specification with derived input condition $x_0 \neq \text{nil}$:

Hypotheses: h1. a_1=b

h2. Bag:z_1=Add:<a_2, Bag:x_2>

h3. Ordered:z_1

h4. Cons:<b,z_1> = z_0

Variables: $\{a_0, x_0, a_1, a_2, x_2\}$

Goal 1:

<Q> Bag:z_0 = Add:<a_0,Bag:x_0>

| R5 + h4

<Q> Bag:Cons:<b,z_1> = Add:<a_0,Bag:x_0>

| R5 + L8

<Q> Add:<b,Bag:z_1> = Add:<a_0,Bag:x_0>

| R5 + h2

<Q> Add:<a_1,Add:<a_2,Bag:x_2>>=Add:<a_0,Bag:x_0>

P2

where Q is Add:<a_1,Add:<a_2,Bag:x_2>>=Add:<a_0,Bag:x_0>

Goal 2:

<Q3> Ordered:z_0

| R5 + h4

<Q3> Ordered:Cons:<b,z_1>

$\wedge$ R3 + L12, R1

<Q3> b $\leq$ Bag:z_1 <TRUE> Ordered:z_1

| R5 + h1, R5 + h2 P1 + h3

<Q3> $a_1 \leq$ Add:<a_2,Bag:x_2>

$\wedge$ R3 + B3, R1

<Q1> $a_1 \leq a_2$ <Q2> $a_1 \leq$ Bag:x_2

P2 P2

where Q1 is $a_1 \leq a_2$, Q2 is $a_1 \leq$ Bag:x_2 and Q3 is Q1 $\wedge$ Q2

Figure 17: Derivation of Output Conditions for Decomposition Operator in Insert.

$$\text{Decompose}:\langle a_0,x_0\rangle \equiv \text{if}\ a_0 \leq \text{First}:x_0 \rightarrow \langle a_0,\langle \text{First}:x_0,\text{Rest}:x_0\rangle\rangle \ [\!]\ a_0 \geq \text{First}:x_0 \rightarrow \langle \text{First}:x_0,\langle a_0,\text{Rest}:x_0\rangle\rangle\ \text{fi}.$$

5. Determine the guard.

The input condition on σ_E is $x_0 \neq \text{nil}$ thus $q:x_0 \Leftrightarrow x_0 = \text{nil}$. To verify that q is defined whenever the input condition holds we easily prove that

$$\forall \langle a,x\rangle \in \mathbb{N} \times \text{LIST}(\mathbb{N})[\ \text{Ordered}:x \Rightarrow \text{Defined}:(x = \text{nil})\ ].$$

6. Construct the primitive operator.

The primitive operator must satisfy

$$h:\langle a_0,x_0\rangle = z_0 \text{ such that } x_0 = \text{nil} \wedge \text{Ordered}:x_0 \Rightarrow \text{Bag}:z_0 = \text{Add}:\langle a_0,\text{Bag}:x_0\rangle \wedge \text{Ordered}:z_0$$
$$\text{where } h: \mathbb{N} \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

It is easily shown that the operator Cons satisfies this specification.

7. Construction of a new input condition.

Since Cons has been found to satisfy the specification h this step is bypassed.

8. Final assembly of the divide and conquer algorithm.

Putting together the operators derived above we obtain

$$\text{Insert}:\langle a_0,x_0\rangle \equiv \text{if}\ x_0 = \text{nil} \rightarrow \text{Cons}:\langle a_0,x_0\rangle \ [\!]\ x_0 \neq \text{nil} \rightarrow \text{Cons}\cdot(\text{Id} \times \text{Insert})\cdot\text{Decompose}:\langle a_0,x_0\rangle\ \text{fi}.$$

with derived input condition TRUE. The completed insertion sort algorithm Isort is given in Figure 18.

```
Isort:x ≡ if
            x = nil  →  x  []
            x ≠ nil  →  Insert·(Id ⨯ Isort)·[First,Rest]:x
          fi


Insert:<a,x> ≡ if
                 x = nil  →  Cons:<a,x>  []
                 x ≠ nil  →  Cons·(Id ⨯ Insert)·Decompose:x
               fi


Decompose:<a0,x0> ≡ if
                      a0 ≤ First:x0  →  <a0,<First:x0,Rest:x0>>  []
                      a0 ≥ First:x0  →  <First:x0,<a0,Rest:x0>>
                    fi
```

Figure 18. Complete Insertion Sort Algorithm

4.2 Synthesis of a Merge Sort Algorithm

4.2.1 Synthesis of Msort

Again we start with specification

$$\text{Sort:}x = z \text{ such that } \text{Bag:}x = \text{Bag:}z \wedge \text{Ordered:}z$$
$$\text{where Sort:LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The synthesis of a mergesort (and a quicksort) distinguishes itself from Isort and Ssort immediately in the choice of signature.

1. Choose a sort set and signature.

As before LIST(ℕ) is the only choice for recurrent type on both the input and output domains. Suppose however we choose the algebra

$B = \langle\{LIST(\mathbb{N})\},\{Append\}\rangle$ which has sort set $S = \{\hat{S}\}$ and a simple S-sorted signature of type $\langle\hat{S}\hat{S},\hat{S}\rangle$. I.e., $Append: B_{\hat{S}} \times B_{\hat{S}} \rightarrow B_{\hat{S}}$.

2. Determine the component problems.

Let $E_{\hat{S}} = LIST(\mathbb{N})$, $T_{\hat{S}} = LIST(\mathbb{N})$, $J_{\hat{S}} = TRUE$, and $P_{\hat{S}}: \langle x,z\rangle \Leftrightarrow Bag:x = Bag:z \wedge Ordered:z$. There are no auxiliary problems.

3. Determine a well-founded ordering on $E_{\hat{S}}$.

Again we define $x_0 \succ x_1$ by $Length:x_0 > Length:x_1$ as an appropriate well-founded ordering by Proposition 1.

ET 4. Construct E then T.

Mergesort differs from quicksort in following track ET rather than TE. In other words mergesort, like insertion sort, uses a simple decomposition operator and a complex composition operator whereas the reverse holds for quicksort.

ET 4.1 Construct E.

After instantiating and simplifying the schematic specification in step ET 4.1 of the synthesis method we obtain

$$\sigma_E: x_0 = \langle x_1, x_2\rangle \text{ such that } Length:x_0 > Length:x_1 \wedge Length:x_0 > Length:x_2$$
$$\text{where } \sigma_E: LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}) \times LIST(\mathbb{N}).$$

In Example 3.1.1 we showed that the primitive operator Listsplit satisfies this specification with derived input condition $Length:x_0 > 1$.

ET 4.2 Construct T.

ET 4.2.1 Derive output conditions for σ_T.

The output condition of the composition operator is found by deriving a $\{z_0,z_1,z_2\}$-precondition of

$\forall \langle z_0,z_1,z_2\rangle \in LIST(\mathbb{N}) \times LIST(\mathbb{N}) \times LIST(\mathbb{N})$
$\forall \langle x_0,x_1,x_2\rangle \in LIST(\mathbb{N}) \times LIST(\mathbb{N}) \times LIST(\mathbb{N})$
$[Length:x_1 = Length:x_0 \text{ div } 2 \wedge Length:x_2 = (1 + Length:x_0) \text{ div } 2 \wedge$
$Append:x_0 = \langle x_1,x_2\rangle \wedge Length:x_0 > Length:x_1 \wedge Length: x_0 > Length:x_2 \wedge$

$\text{Bag}:x_1 = \text{Bag}:z_1 \wedge \text{Ordered}:z_1 \wedge \text{Bag}:x_2 = \text{Bag}:z_2 \wedge \text{Ordered}:z_2$
$\Rightarrow \text{Bag}:x_0 = \text{Bag}:z_0 \wedge \text{Ordered}:z_0]$.

The precondition

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow (\text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2 \rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0)$$

is derived in Figure 19.

ET 4.2.2 Construct T.

Instantiating the above output condition in the schema ET 4.2.2 we obtain the specification

$$\sigma_T:\langle z_1, z_2 \rangle = z_0 \text{ such that } \text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow$$
$$\text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2 \rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$$
$$\text{where } \sigma_T:\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

In Section 4.2.2 we derive a program called Merge satisfying this specification.

5. Determine the guards.

The guard $\sim q$ is simply $\text{Length}:x_0 > 1$ - the input condition on σ_E. Negating, we obtain $q:x \Leftrightarrow \text{Length}:x \leq 1$. It is easily shown that q is defined on all legal inputs.

6. Construct the primitive operator.

The primitive operator has specification

$$h:x = z \text{ such that } \text{Length}:x \leq 1 \Rightarrow \text{Bag}:x = \text{Bag}:z \wedge \text{Ordered}:z$$
$$\text{where } h:\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The identity operator Id is easily shown to satisfy this specification.

7. Construct a new input condition.

This step is skipped since an operator has been found which satisfies the specification h.

8. Construction of the divide and conquer algorithm.

Hypotheses:
1. $\text{Append}:\langle x_1, x_2\rangle = x_0$
2. $\text{Length}:x_1 = \text{Length}:x_0 \text{ div } 2$
3. $\text{Length}:x_2 = (1 + \text{Length}:x_0) \text{ div } 2$
4. $\text{Length}:x_0 > \text{Length}:x_1$
5. $\text{Length}:x_0 > \text{Length}:x_2$
6. $\text{Bag}:x_1 = \text{Bag}:z_1$
7. $\text{Ordered}:z_1$
8. $\text{Bag}:x_2 = \text{Bag}:z_2$
9. $\text{Ordered}:z_2$

Variables: $\{z_0, z_1, z_2\}$

Goal 1:

$$\langle Q\rangle \quad \text{Bag}:x_0 = \text{Bag}:z_0$$
$$\Big| \; R5 + h1$$
$$\langle Q\rangle \quad \text{Bag}:\text{Append}:\langle x_1, x_2\rangle = \text{Bag}:z_0$$
$$\Big| \; R5 + L9$$
$$\langle Q\rangle \quad \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle = \text{Bag}:z_0$$
$$\Big| \; R5 + h6,\ R5 + h8$$
$$\langle Q\rangle \quad \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle = \text{Bag}:z_0$$
$$P2$$

where Q is $\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle = \text{Bag}:z_0$

Goal 2:

$$\langle \text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Ordered}:z_0\rangle \; \text{Ordered}:z_0$$
$$P2$$

Figure 19: Derivation of output conditions for Merge.

Assembling the operators derived above we obtain the program

$$\text{Msort}:x \equiv \text{if}$$
$$\text{Length}:x \leq 1 \rightarrow x \; []$$
$$\text{Length}:x > 1 \rightarrow \text{Merge} \cdot (\text{Msort} \times \text{Msort}) \cdot \text{Listsplit}:x$$
$$\text{fi}$$

The derived input condition on Msort is TRUE.

4.2.2 Synthesis of Merge

In the previous section we derived the following specification for a composition operator which will be called Merge.

$$\text{Merge}:\langle x_0,x'_0\rangle = z_0 \text{ such that } \text{Ordered}:x_0 \wedge \text{Ordered } x'_0 \Rightarrow$$
$$\text{Union}:\langle \text{Bag}:x_0,\text{Bag}:x'_0\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$$
$$\text{where Merge}:\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

A divide and conquer algorithm for Merge is synthesized as follows.

1. Determine sort set and signature.

There is only one choice of recurrent type in both the input and output domains - namely LIST(IN). As before we obtain sort set $S = \{c,\hat{s}\}$ and signature Σ of type $\langle c\hat{s},\hat{s}\rangle$ from the algebra $A = \langle\{\mathbb{N},\text{LIST}(\mathbb{N})\},\{\text{Cons}\}\rangle$.

2. Determine the component problems.

Let

$$E_{\hat{s}} = \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$$
$$T_{\hat{s}} = \text{LIST}(\mathbb{N})$$
$$J_{\hat{s}}:\langle x_0,x'_0\rangle \Leftrightarrow \text{Ordered}:x_0 \wedge \text{Ordered}:x'_0$$
$$P_{\hat{s}}:\langle\langle x_0,x'_0\rangle,z_0\rangle \Leftrightarrow \text{Union}:\langle \text{Bag}:x_0,\text{Bag}:x'_0\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$$
$$E_c = A_c = \mathbb{N}$$
$$T_c = A_c = \mathbb{N}$$

again we find $\text{Id}:E_c \rightarrow T_c$ so let

$$J_c:x \Leftrightarrow \text{TRUE}$$
$$P_c:\langle x,z\rangle \Leftrightarrow x = z$$
$$f_c \equiv \text{Id}.$$

3. Determine a well-founded ordering on $E_{\hat{s}}$.

Since $E_{\hat{s}} = \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$ we can construct a well-founded ordering by seeking a mapping from $\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$ to $\mathbb{N} \times \mathbb{N}$. The function product Length $\times$ Length suffices so we define

$$\langle x_0,x'_0\rangle \succ \langle x_1,x'_1\rangle \text{ iff } \text{Length} \times \text{Length}:\langle x_0,x'_0\rangle >_2 \text{Length} \times \text{Length}:\langle x_1,x'_1\rangle$$

where $\langle i_0,i'_0\rangle >_2 \langle i_1,i'_1\rangle$ iff $i_0>i_1$ or $(i_0=i_1 \wedge i'_0>i'_1)$.

4. Construct E and T.

For Merge we construct σ_T then σ_E.

TE 4.1 Construct T.

The composition operator has specification

$$\sigma_T:\langle b,z_1\rangle = z_0 \text{ such that TRUE}$$
$$\text{where } \sigma_T: \mathbb{N} \times LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$$

The primitive operator Cons can be shown to satisfy σ_T.

TE 4.2 Construct E

TE 4.2.1 Derive output specifications for σ_E.

To obtain output conditions for the decomposition operator we derive a $\{x_0,x'_0,a,x_1,x'_1\}$-precondition of

$\forall\langle\langle x_0,x'_0\rangle,a,\langle x_1,x'_1\rangle\rangle \in (LIST(\mathbb{N}) \times LIST(\mathbb{N})) \times \mathbb{N} \times (LIST(\mathbb{N}) \times LIST(\mathbb{N}))$
$\forall\langle z_0,b,z_1\rangle \in LIST(\mathbb{N}) \times \mathbb{N} \times LIST(\mathbb{N})$
$[a = b \wedge Ordered{:}x_1 \wedge Ordered{:}x'_1 \wedge Union{:}\langle Bag{:}x_1, Bag{:}x'_1\rangle = Bag{:}z_1 \wedge$
$Ordered{:}z_1 \wedge Cons{:}\langle b,z_1\rangle = z_0 \Rightarrow Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Bag{:}z_0 \wedge Ordered{:}z_0]$.

The derivation in Figures 21a and 21b yields the precondition

$$Ordered{:}x_1 \wedge Ordered{:}x'_1 \Rightarrow$$
$Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Add{:}\langle a, Union{:}\langle Bag{:}x_1, Bag{:}x'_1\rangle\rangle \wedge a \leq Bag{:}x_1 \wedge a \leq Bag{:}x'_1$.

TE 4.2.2 Construct E.

The decomposition operator σ_E has specification

$\sigma_E:\langle x_0,x'_0\rangle = \langle a,\langle x_1,x'_1\rangle\rangle$ such that $Ordered{:}x_0 \wedge Ordered{:}x'_0 \Rightarrow$
$Ordered{:}x_1 \wedge Ordered{:}x'_1 \wedge Length \times Length{:}\langle x_0,x'_0\rangle >_2 Length \times Length{:}\langle x_1,x'_1\rangle$
$\wedge Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Add{:}\langle a, Union{:}\langle Bag{:}x_1, Bag{:}x'_1\rangle\rangle \wedge a \leq Bag{:}x_1 \wedge a \leq Bag{:}x'_1$
where $\sigma_E: LIST(\mathbb{N}) \times LIST(\mathbb{N}) \rightarrow \mathbb{N} \times (LIST(\mathbb{N}) \times LIST(\mathbb{N}))$.

Hypotheses: 1. $a = b$

2. $Ordered{:}x_1$

3. $Ordered{:}x'_1$

4. $Union{:}\langle Bag{:}x_1, Bag{:}x'_1\rangle = Bag{:}z_1$

5. $Ordered{:}z_1$

6. $Cons{:}\langle b, z_1\rangle = z_0$

Variables: $\{x_0, x'_0, a, x_1, x'_1\}$

Goal 1: $\langle Q1\rangle \quad Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Bag{:}z_0$

| R5 + h6

$\langle Q1\rangle \quad Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Bag{:}Cons{:}\langle a, z_1\rangle$

| R5 + L8

$\langle Q1\rangle \quad Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Add{:}\langle a, Bag{:}z_1\rangle$

| R5 + h4

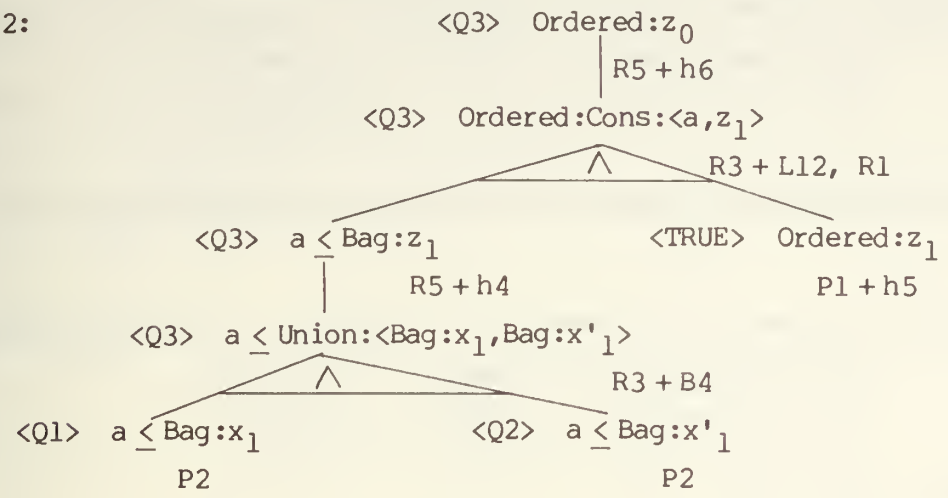
$\langle Q1\rangle \quad Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Add{:}\langle a, Union{:}\langle Bag{:}x_1, Bag{:}x'_1\rangle\rangle$

P2

where Q1 is
$Ordered{:}x_1 \wedge Ordered{:}x'_1 \Rightarrow Union{:}\langle Bag{:}x_0, Bag{:}x'_0\rangle = Add{:}\langle a, Union{:}\langle Bag{:}x_1, Bag{:}x'_1$

Figure 20a: Deriving output specification for the composition operator in Merge

Goal 2:



where Q1 is Ordered:x_1 ∧ Ordered:x'_1 ⇒ a ≤ Bag:x_1,
Q2 is Ordered:x_1 ∧ Ordered:x'_1 ⇒ a ≤ Bag:x'_1,
and Q3 is Ordered:x_1 ∧ Ordered:x'_1 ⇒ Q1 ∧ Q2

Figure 20b: Deriving output specification for the composition operator in Merge

A simple conditional program called Decompose can be constructed satisfying this specification with derived input condition $x_0 \neq nil \wedge x'_0 \neq nil$

Decompose:$\langle x_0, x'_0 \rangle \equiv$ if
$First{:}x_0 \leq First{:}x'_0 \rightarrow \langle First{:}x_0, \langle Rest{:}x_0, x'_0 \rangle \rangle$ ▯
$First{:}x_0 \geq First{:}x'_0 \rightarrow \langle First{:}x'_0, \langle x_0, Rest{:}x'_0 \rangle \rangle$
fi

5. Determine the guard.

The derived input condition on Decompose is

$$x_0 \neq nil \wedge x'_0 \neq nil$$

thus we define $q{:}\langle x_0, x'_0 \rangle$ to be $x_0 = nil \vee x'_0 = nil$. We can easily verify that q is defined for all pairs of input lists.

6. Construct the primitive operator.

We create the specification

$$h{:}\langle x_0, x'_0\rangle = z_0 \text{ such that } \mathrm{Ordered}{:}x_0 \wedge \mathrm{Ordered}{:}x'_0 \wedge (x_0 = \mathrm{nil} \vee x'_0 = \mathrm{nil}) \Rightarrow \mathrm{Union}{:}\langle \mathrm{Bag}{:}x_0, \mathrm{Bag}{:}x'_0\rangle = \mathrm{Bag}{:}z_0 \wedge \mathrm{Ordered}{:}z_0$$

where $h{:}\mathrm{LIST}(\mathbb{N}) \times \mathrm{LIST}(\mathbb{N}) \rightarrow \mathrm{LIST}(\mathbb{N})$.

We treat the disjunctive input condition by splitting the specification into two cases:

$$h_0{:}\langle x_0, x'_0\rangle = z_0 \text{ such that } \mathrm{Ordered}{:}x_0 \wedge \mathrm{Ordered}{:}x'_0 \wedge x_0 = \mathrm{nil} \Rightarrow \mathrm{Union}{:}\langle \mathrm{Bag}{:}x_0, \mathrm{Bag}{:}x'_0\rangle = \mathrm{Bag}{:}z_0 \wedge \mathrm{Ordered}{:}z_0$$

where $h_0{:}\mathrm{LIST}(\mathbb{N}) \times \mathrm{LIST}(\mathbb{N}) \rightarrow \mathrm{LIST}(\mathbb{N})$

and

$$h'_0{:}\langle x_0, x'_0\rangle = z_0 \text{ such that } \mathrm{Ordered}{:}x_0 \wedge \mathrm{Ordered}{:}x'_0 \wedge x'_0 = \mathrm{nil} \Rightarrow \mathrm{Union}{:}\langle \mathrm{Bag}{:}x_0, \mathrm{Bag}{:}x'_0\rangle = \mathrm{Bag}{:}z_0 \wedge \mathrm{Ordered}{:}z_0$$

where $h'_0{:}\mathrm{LIST}(\mathbb{N}) \times \mathrm{LIST}(\mathbb{N}) \rightarrow \mathrm{LIST}(\mathbb{N})$

and synthesizing separate programs for them. The selector functions $\underset{\sim}{2}$ and $\underset{\sim}{1}$ are easily shown to satisfy h_0 and h'_0 respectively.

7. Create new input conditions.

This step is skipped since operators have been derived which satisfy the specifications h_0 and h'_0.

8. Assembly of a divide and conquer algorithm.

Putting together the operators derived above we obtain

$$\begin{aligned}\mathrm{Merge}{:}\langle x_0, x'_0\rangle \equiv\ & \mathrm{if}\\ & \quad x_0 = \mathrm{nil} \rightarrow x'_0 \ \square\\ & \quad x'_0 = \mathrm{nil} \rightarrow x_0 \ \square\\ & \quad x_0 \neq \mathrm{nil} \wedge x'_0 \neq \mathrm{nil} \rightarrow \mathrm{Cons}\cdot(\mathrm{Id} \times \mathrm{Merge})\cdot\mathrm{Decompose}{:}\langle x_0, x'_0\rangle\\ & \mathrm{fi}\end{aligned}$$

The derived input condition for Merge is TRUE. The complete Merge sort program

```
Msort:x ≡ if
            Length:x ≤ 1  →  x  []
            Length:x > 1  →  Merge·(Msort ╳ Msort)·Listsplit:x
          fi

Merge:<x1,x2> ≡ if
                  x1 = nil  →  x2  []
                  x2 = nil  →  x1  []
                  x1 ≠ nil ∧ x2 ≠ nil  →  Cons·(Id ╳ Merge)·Decompose:<x1,x2>
                fi

Decompose:<x1,x2> ≡ if
                       First:x1 ≤ First:x2  →  <First:x1,<Rest:x1,x2>>  []
                       First:x1 ≥ First:x2  →  <First:x2,<x1,Rest:x2>>
                     fi
```

Figure 21: Complete Merge Sort Algorithm.

4.3 Synthesis of a Quick Sort Algorithm

4.3.1 Synthesis of Qsort

The synthesis of a quicksort proceeds as with Mergesort diverging only at step 4.

1,2,3. Determine a sort set, signature, component problems, and a well-founded ordering.

As in Mergesort let

$S = \{\hat{S}\}$,

$\sum$ be a simple S-sorted signature of type $\{\hat{S}\hat{S},\hat{S}\}$,

$E_{\hat{S}} = LIST(\mathbb{N})$,

$T_{\hat{S}} = LIST(\mathbb{N})$,

$J_{\hat{S}}:x \Leftrightarrow TRUE$, and

$$P_S:\langle x,z\rangle \iff \text{Bag}:x = \text{Bag}:z \wedge \text{Ordered}:z.$$

Define $x_0 \succ x_1$ iff $\text{Length}:x_0 > \text{Length}:x_1$.

4. Construct decomposition and composition operators.

In Msort we chose a simple decomposition operator, in Qsort we choose a simple composition operator.

TE 4.1 Construct T.

The composition operator has partial specification

$$\sigma_T:\langle z_1,z_2\rangle = z_0 \text{ such that TRUE}$$
$$\text{where } \sigma_T:\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}).$$

The operator Append satisfies σ_T with derived input condition TRUE.

TE 4.2 Construct E.

TE 4.2.1 Derive output conditions for σ_E.

We seek a $\{x_0,x_1,x_2\}$-precondition of

$\forall\langle x_0,x_1,x_2\rangle \in \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$
$\forall\langle z_0,z_1,z_2\rangle \in \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$
$[\text{Bag}:x_1 = \text{Bag}:z_1 \wedge \text{Ordered}:z_1 \wedge \text{Bag}:x_2 = \text{Bag}:z_2 \wedge \text{Ordered}:z_2 \wedge$
$\text{Append}:\langle z_1,z_2\rangle = z_0 \Rightarrow \text{Bag}:x_0 = \text{Bag}:z_0 \wedge \text{Ordered}:z_0]$

and in Figure 22 we derive

$$\text{Bag}:x_1 \leq \text{Bag}:x_2 \wedge \text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:x_1,\text{Bag}:x_2\rangle.$$

TE 4.2.2 Construct σ_E.

Using previously derived results we construct by instantiation the specification

$$\sigma_E:x_0 = \langle x_1,x_2\rangle \text{ such that } \text{Length}:x_0 > \text{Length}:x_1 \wedge \text{Length}:x_0 > \text{Length}:x_2 \wedge$$
$$\text{Bag}:x_1 \leq \text{Bag}:x_2 \wedge \text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:x_1,\text{Bag}:x_2\rangle$$
$$\text{where } \sigma_E:\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}).$$

Hypotheses: h1. $Bag{:}x_1 = Bag{:}z_1$

h2. $Ordered{:}z_1$

h3. $Bag{:}x_2 = Bag{:}z_2$

h4. $Ordered{:}z_2$

h5. $Append{:}\langle z_1,z_2\rangle = z_0$

Variables: $\{x_0,x_1,x_2\}$

Goal 1:

<Q1> $Bag{:}x_0 = Bag{:}z_0$

| R5 + h5

<Q1> $Bag{:}x_0 = Bag{:}Append{:}\langle z_1,z_2\rangle$

| R5 + L9

<Q1> $Bag{:}x_0 = Union{:}\langle Bag{:}z_1,Bag{:}z_2\rangle$

| R5 + h1, R5 + h3

<Q1> $Bag{:}x_0 = Union{:}\langle Bag{:}x_1,Bag{:}x_2\rangle$

P2

where Q1 is $Bag{:}x_0 = Union{:}\langle Bag{:}x_1,Bag{:}x_2\rangle$

Goal 2:

<Q2> $Ordered{:}z_0$

| R5 + h5

<Q2> $Ordered{:}Append{:}\langle z_1,z_2\rangle$

$\wedge$ R3 + L13, R1, R1

<TRUE> $Ordered{:}z_1$ <Q2> $Bag{:}z_1 \leq Bag{:}z_2$ <TRUE> $Ordered{:}z_2$

P1 + h2 | R5 + h1, R5 + h3 P1 + h4

<Q2> $Bag{:}x_1 \leq Bag{:}x_2$

P2

where Q2 is $Bag{:}x_1 \leq Bag{:}x_2$

Figure 22: Derivation of output conditions for the decomposition operator of Qsort

---

This is an incomplete specification for the well-known partitioning operator of a quicksort. In the following section we synthesize an operator called Part satisfying this specification with derived input condition $Length{:}x_0>1$.

5. Determine the guard.

The guard $\sim q{:}x_0$ is simply the derived input condition on σ_E so $\sim q{:}x_0 \Leftrightarrow$ $Length{:}x_0 > 1$ and $q{:}x_0 \Leftrightarrow Length{:}x_0 \leq 1$. It is easily verified that q is defined on all legal inputs.

6. Construct the primitive operator.

We construct specification

$$h{:}x_0 = z_0 \text{ such that } Length{:}x_0 \leq 1 \Rightarrow Bag{:}x_0 = Bag{:}z_0 \wedge Ordered{:}z_0$$
$$\text{where } h{:}LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$$

and can easily show that Id satisfies it.

7. Derive a new input condition.

Since Id satisfies specification h this step is bypassed.

8. Assemble divide and conquer algorithm.

The operators constructed above compose to form the following program:

$Qsort{:}x \equiv$ if
$\quad Length{:}x \leq 1 \rightarrow x \;[]$
$\quad Length{:}x > 1 \rightarrow Append \cdot (Qsort \times Qsort) \cdot Part{:}x$
fi

The derived input condition on Qsort is TRUE.

4.3.3 Synthesis of Partition

In the previous section we derived the specification

$$Part{:}x_0 = \langle x_1, x_2 \rangle \text{ such that } Length{:}x_0 > Length{:}x_1 \wedge Length{:}x_0 > Length{:}x_2 \wedge$$
$$Bag{:}x_1 \leq Bag{:}x_2 \wedge Bag{:}x_0 = Union{:}\langle Bag{:}x_1, Bag{:}x_2 \rangle$$
$$\text{where } Part{:}LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}) \times LIST(\mathbb{N}).$$

This specification can be transformed to a somewhat simpler form using the equivalence

$$w_1 \leq w_2 \Leftrightarrow \exists i [w_1 \leq i \wedge i \leq w_2]$$

where w_1 and w_2 are variables over BAGS(IN) and i varies over IN. Thus we get

$$\text{Part1}:x_0 = \langle x_1, x_2\rangle \text{ such that } \exists\, i[\text{Bag}:x_1 \leq i \wedge i \leq \text{Bag}:x_2] \wedge$$
$$\text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle \wedge \text{Length}:x_0 > \text{Length}:x_1 \wedge \text{Length}:x_0 > \text{Length}:x_2$$
$$\text{where Part1}:\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}).$$

We can further transform this to

$$\text{Part2}:x_0 = \langle b, \langle z_1, z_2\rangle\rangle \text{ such that } \text{Bag}:z_1 \leq b \wedge b \leq \text{Bag}:z_2 \wedge$$
$$\text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle \wedge \text{Length}:x_0 > \text{Length}:z_1 \wedge \text{Length}:x_0 > \text{Length}:z_2$$
$$\text{where Part2}:\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))$$

Noting that $\underset{\sim}{2}\cdot\text{Part2}:x_0$ satisfies the specification for Part we now synthesize a divide and conquer algorithm for Part2.

1. Determine a sort set and signature.

We choose LIST(ℕ) as recurrent type on the input and output domains and select the algebra $A = \langle\{\mathbb{N}, \text{LIST}(\mathbb{N})\}, \{\text{Cons}\}\rangle$ to give us the sort set $S = \{c, \hat{s}\}$ and signature Σ of type $\langle c\hat{s}, \hat{s}\rangle$.

2. Determine the component problems.

As in the synthesis of Select, Insert, and Merge let

$E_{\hat{s}} = \text{LIST}(\mathbb{N})$

$T_{\hat{s}} = \mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))$

$J_{\hat{s}}:x \Leftrightarrow \text{TRUE}$

$P_{\hat{s}}:\langle x_0, \langle b, \langle z_1, z_2\rangle\rangle\rangle \Leftrightarrow$

$\quad \text{Bag}:z_1 \leq b \wedge b \leq \text{Bag}:z_2 \wedge$

$\quad \text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle \wedge$

$\quad \text{Length}:x_0 > \text{Length}:z_1 \wedge \text{Length}:x_0 > \text{Length}:z_2.$

$E_c = \mathbb{N}$, and

$T_c = \mathbb{N}.$

We seek a function f_s which maps E_c to T_c and find Id, so

$J_c:x \Leftrightarrow \text{TRUE}$

$P_c:\langle x, z\rangle \Leftrightarrow x = z.$

3. Determine a well-founded ordering on $E_{\hat{s}}$.

Again define $x_0 \succ x_1$ by $\text{Length}{:}x_0 > \text{Length}{:}x_1$.

4. Construct E then T.

ET 4.1 Construct E.

We construct a simple decomposition operator according to the incomplete specification

$$\sigma_E{:}x_0 = \langle a, x_1\rangle \text{ such that } \text{Length}{:}x_0 > \text{Length}{:}x_1$$
$$\text{where } \sigma_E{:}\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N}).$$

In the synthesis of Select we showed that [First,Rest] satisfies the same specification with derived input condition $x_0 \neq \text{nil}$.

ET 4.2 Construct T.

ET 4.2.1 Derive output conditions for σ_T.

The composition operator's output conditions are found by deriving a $\{b, c_1, z_1, z'_1, c_0, z_0, z'_0\}$-precondition of

$\forall \langle c_0, \langle z_0, z'_0\rangle\rangle \in \mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))$
$\forall \langle b, \langle c_1, \langle z_1, z'_1\rangle\rangle\rangle \in \mathbb{N} \times (\mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})))$
$\forall \langle x_0, a, x_1\rangle \in \text{LIST}(\mathbb{N}) \times \text{LIST}\,\mathbb{N} \times \text{LIST}(\mathbb{N})$
$[\,[\text{First,Rest}]{:}x_0 = \langle a, x_1\rangle \wedge \text{Bag}{:}z_1 \leq c_1 \wedge c_1 \leq \text{Bag}{:}z'_1 \wedge$
$\text{Bag}{:}x_1 = \text{Union}{:}\langle \text{Bag}{:}z_1, \text{Bag}{:}z'_1\rangle \wedge \text{Length}{:}x_1 > \text{Length}{:}z_1 \wedge \text{Length}{:}x_1 > \text{Length}{:}z'_1 \wedge a = b$
$\Rightarrow \text{Bag}{:}z_0 \leq c_0 \wedge c_0 \leq \text{Bag}{:}z'_0 \wedge \text{Bag}{:}x_0 = \text{Union}{:}\langle \text{Bag}{:}z_0, \text{Bag}{:}z'_0\rangle \wedge$
$\text{Length}{:}x_0 > \text{Length}{:}z_0 \wedge \text{Length}{:}x_0 > \text{Length}{:}z'_0\,]$.

In Figures 24a and 24b we derive the precondition

$$\text{Bag}{:}z_1 \leq c_1 \wedge c_1 \leq \text{Bag}{:}z'_1 \Rightarrow 1 + \text{Length}{:}z_1 + \text{Length}{:}z'_1 > \text{Length}{:}z_0 \wedge$$
$$1 + \text{Length}{:}z_1 + \text{Length}{:}z'_1 > \text{Length}{:}z'_0 \wedge$$
$$\text{Add}{:}\langle b, \text{Union}{:}\langle \text{Bag}{:}z_1, \text{Bag}{:}z'_1\rangle\rangle = \text{Union}{:}\langle \text{Bag}{:}z_0, \text{Bag}{:}z'_0\rangle \wedge$$
$$\text{Bag}{:}z_0 \leq c_0 \wedge c_0 \leq \text{Bag}{:}z_0.$$

ET 4.2.2 Construct T.

Hypotheses: 1. $[First,Rest]:x_0 = \langle a,x_1 \rangle$

2. $Bag:z_1 \leq c_1$

3. $c_1 \leq Bag:z'_1$

4. $Bag:x_1 = Union:\langle Bag:z_1, Bag:z'_1 \rangle$

5. $Length:x_1 > Length:z_1$

6. $Length:x_1 > Length:z'_1$

7. $a = b$

Variables: $\{b,c_1,z_1,z'_1,c_0,z_0,z'_0\}$

Goal 1:

$$\langle Q1 \rangle \quad Bag:x_0 = Union:\langle Bag:z_0, Bag:z'_0 \rangle$$

$$\mid \quad R6 + L7 + h1$$

$$\langle Q1 \rangle \quad Bag:Cons:\langle a,x_1 \rangle = Union:\langle Bag:z_0, Bag:z'_0 \rangle$$

$$\mid \quad R5 + L8$$

$$\langle Q1 \rangle \quad Add:\langle a,Bag:x_1 \rangle = Union:\langle Bag:z_0, Bag:z'_0 \rangle$$

$$\mid \quad R5 + h4$$

$$\langle Q1 \rangle \quad Add:\langle b,Union:\langle Bag:z_1, Bag:z'_1 \rangle \rangle = Union:\langle Bag:z_0, Bag:z'_0 \rangle$$

P2

where Q1 is

$$Bag:z_1 \leq c_1 \wedge c_1 \leq Bag:z'_1 \Rightarrow Add:\langle b,Union:\langle Bag:z_1, Bag:z'_1 \rangle \rangle = Union:\langle Bag:z_0, Bag:z'_0 \rangle$$

Goal 2:

$$\langle Q2 \rangle \quad Bag:z_0 \leq c_0$$

P2

where Q2 is $Bag:z_1 \leq c_1 \wedge c_1 \leq Bag:z'_1 \Rightarrow Bag:z_0 \leq c_0$

Goal 3:

$$\langle Q3 \rangle \quad c_0 \leq Bag:z'_0$$

P2

where Q3 is $Bag:z_1 \leq c_1 \wedge c_1 \leq Bag:z'_1 \Rightarrow c_0 \leq Bag:z'_0$

Figure 23a: Deriving output conditions for the decomposition operator in Part2.

The specification for the composition operator σ_T is

σ_T:<b,<c_1,<z_1,z'_1>>> = <c_0,<z_0,z'_0>> such that Bag:$z_1 \leq c_1 \wedge c_1 \leq$ Bag:z'_1

$\Rightarrow$ (Add:<b,Union:<Bag:z_1,Bag:z_2>> = Union:<Bag:z_0,Bag:z'_0> $\wedge$

1 + Length:z_1 + Length:z'_1>Length:z_0 $\wedge$

1 + Length:z_1 + Length:z'_1>Length:z_0 $\wedge$

Bag:$z_0 \leq c_0 \wedge c_0 \leq$ Bag:z_0

where σ_T: $\mathbb{N} \times (\mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))) \rightarrow \mathbb{N} \times (\text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))$

A simple conditional program can be derived satisfying this specification with derived input condition TRUE:

Goal 4:

<Q4> Length:x_0 > Length:z_0

| R6 + L7 + h1

<Q4> Length:Cons:<a,x_1> > Length:z_0

| R5 + L15

<Q4> 1 + Length:x_1 > Length:z_0

| R6 + L19 + h4

<Q4> 1 + Card:(Union:<Bag:z_1,Bag:z'_1>) > Length:z_0

| R5 + B5

<Q4> 1 + Card:Bag:z_1 + Card:Bag:z'_1 > Length:z_0

| R5 + L18

<Q4> 1 + Length:z_1 + Length:z'_1 > Length:z_0

P2

where Q4 is Bag:$z_1 \leq c_1 \wedge c_1 \leq$ Bag:z'_1 $\Rightarrow$ 1 + Length:z_1 + Length:z'_1 > Length:z_0

Goal 5: <Q5> Length:x_0 > Length:z'_0

(Derivation similar to the derivation of Goal 4)

where Q5 is Bag:$z_1 \leq c_1 \wedge c_1 \leq$ Bag:z'_1 $\Rightarrow$ 1 + Length:z_1 + Length:z'_1 > Length:z'_0

Figure 23b: Deriving output conditions for the decomposition operator in Part2.

Compose:<b,<c,<z,z'>>> ≡ if

$b \leq c \rightarrow$ <c,Cons:<b,z>,z'> []

$b \geq c \rightarrow$ <c,z,Cons:<b,z'>>

fi.

5. Determine the guard.

We take as ~q the derived input condition $x \neq nil$ of σ_E, thus $q:x \Leftrightarrow x = nil$. It is easily shown that q is defined on all legal inputs.

6. Construct the primitive operator.

We construct the specification

$$h:x = \langle c,\langle z,z'\rangle\rangle \text{ such that } x = nil \Rightarrow Bag:z \leq c \wedge c \leq Bag:z' \wedge Bag:x = Union:\langle Bag:z, Bag:z'\rangle \wedge Length:x > Length:z_1 \wedge Length:x > Length:z'_1$$

where $h:LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}) \times LIST(\mathbb{N})$.

As in the synthesis of Select we can show that this specification is unsatisfiable.

7. Derive a new input condition.

We form a new input condition (because of the unsatisfiablility of h) by simplifying

$$(TRUE \wedge x_0 \neq nil) \vee (TRUE \wedge x = nil \wedge FALSE)$$

to $x_0 \neq nil$. We let $J_S:x_0 \Leftrightarrow x_0 \neq nil$ and return to step 4 in order to redo the synthesis.

4' Construct E and T.

ET 4.1' Construct E.

The new specification for σ_E is

$$\sigma_E:x_0 = \langle a,x_1\rangle \text{ such that } x_0 \neq nil \Rightarrow (x_1 \neq nil \wedge Length:x_0 > Length:x_1)$$

where $\sigma_E:LIST(\mathbb{N}) \rightarrow \mathbb{N} \times LIST(\mathbb{N})$.

Since the operator [First,Rest] satisfied the first specification for σ_E it

seems reasonable to try it again. Proposition 4 is used to show that [First,Rest] satisfies σ_E with derived input condition Rest:x $\neq$ nil.

ET 4.2' Construct T.

This step is not affected by the introduction of a new input condition thus our previous synthesis of Compose can be used.

5'. Determine the guards.

This time the derived input condition on σ_E is Rest:x $\neq$ nil so

$$\sim q{:}x \iff \text{Rest:}x \neq \text{nil and } q{:}x \iff \text{Rest:}x = \text{nil}.$$

It is easily verified that q is defined on all legal inputs.

6'. Construct the primitive operator.

The primitive operator has specification

$$h{:}x = \langle c,\langle z,z'\rangle\rangle \text{ such that Rest:}x = \text{nil} \Rightarrow \text{Bag:}z \leq c \wedge c \leq \text{Bag:}z' \wedge$$
$$\text{Bag:}x = \text{Union:}\langle \text{Bag:}z,\text{Bag:}z'\rangle \wedge \text{Length:}x > \text{Length:}z \wedge \text{Length:}x > \text{Length:}z'$$
$$\text{where } h{:}\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times (\text{LIST}(\mathbb{N} \times \text{LIST}(\mathbb{N}))$$

and again we find this to be unsatisfiable. Thus we will need to find a new input condition and return to step 4.

7'. Derive a new input condition.

The new input condition is found by simplifying

$$(x \neq \text{nil} \wedge \text{Rest:}x \neq \text{nil}) \vee (x \neq \text{nil} \wedge \text{Rest:}x = \text{nil} \wedge \text{FALSE})$$

to Length:x > 1. We return again to step 4, letting J_S:x be Length:x > 1.

4''. Construct E and T.

ET 4.1''. Construct E.

The new specification for σ_E is

$\sigma_E{:}x_0 = \langle a, x_1 \rangle$ such that $\text{Length}{:}x_0 > 1 \Rightarrow (\text{Length}{:}x_1 > 1 \wedge \text{Length}{:}x_0 > \text{Length}{:}x_1)$
where $\sigma_E{:}\text{LIST}(\mathbb{N}) \rightarrow \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N})$.

As in step ET 4.1' we use Proposition 4 to match [First, Rest] with σ_E and obtain derived input condition $\text{Length}{:}x_0 > 2$.

This step is the same as ET 4.2 since the change in input condition does not affect the synthesis of σ_T.

5''. Determine the guards.

The derived input condition on σ_E is $\text{Length}{:}x_0 > 2$ so

$$\neg q{:}x \Leftrightarrow \text{Length}{:}x > 2 \text{ and } q{:}x \Leftrightarrow \text{Length}{:}x \leq 2.$$

Again we easily check that q is defined on all legal inputs.

6''. Construct the primitive operator.

The primitive operator has specification

$h{:}x = \langle c, \langle z, z' \rangle \rangle$ such that $\text{Rest}{:}x \neq \text{nil} \wedge \text{Length}{:}x \leq 2 \Rightarrow \text{Bag}{:}z \leq c \wedge c \leq \text{Bag}{:}z' \wedge \text{Bag}{:}x = \text{Union}{:}\langle \text{Bag}{:}z, \text{Bag}{:}z' \rangle \wedge \text{Length}{:}x > \text{Length}{:}z \wedge \text{Length}{:}x > \text{Length}{:}z'$
where $h{:}\text{LIST}(\mathbb{N}) \rightarrow \mathbb{N} \times \text{LIST}(\mathbb{N}) \times \text{LIST}(\mathbb{N}))$.

A simple conditional program can be shown to satisfy this specification:

$h{:}x \equiv$ if
$\quad \text{First}{:}x \leq \text{First}{\cdot}\text{Rest}{:}x \rightarrow \langle \text{First}{:}x, \text{Cons}{:}\langle \text{First}{:}x, \text{nil} \rangle, \text{Rest}{:}x \rangle \; []$
$\quad \text{First}{:}x \geq \text{First}{\cdot}\text{Rest}{:}x \rightarrow \langle \text{First}{\cdot}\text{Rest}{:}x, \text{Rest}{:}x, \text{Cons}{:}\langle \text{First}{:}x, \text{nil} \rangle \rangle$
fi.

7''. Derive a new input condition.

Since an operator was constructed which satisfies the specification for h we can bypass this step.

8''. Assembly of the divide and conquer program.

Putting together all of the operators derived above we obtain

```
Part:x ≡ 2̃·Part2:x

Part2:x ≡ if
            Length:x ≤ 2 → h:x []
            Length:x > 2 → Compose·(Id ⨯ Part2)·[First,Rest]:x
          fi.
```

Length:x>1 is the derived input condition on Part2. The complete Quicksort pro-

---

```
Qsort:x ≡ if
            Length:x ≤ 1 → x []
            Length:x > 1 → Append·(Qsort ⨯ Qsort)·Part:x
          fi

Part:x ≡ 2̃·Part2:x

Part2:x ≡ if
            Length:x ≤ 2 → h:x []
            Length:x > 2 → Compose·(Id ⨯ Part2)·[First,Rest]:x
          fi

h:x ≡ if
        First:x ≤ First·Rest:x → <First:x, Cons:<First:x,nil>, Rest:x> []
        First:x ≥ First·Rest:x → <First·Rest:x, Rest:x, Cons:<First:x,nil>>
      fi

Compose:<a,<b,z1,z2>> ≡ if
                          a ≤ b → <b, Cons:<a,z1>, z2> []
                          a ≥ b → <b, z1, Cons:<a,z2>>
                        fi
```

Figure 24: Complete Quicksort Program

---

gram is listed in Figure 24.

4.4 Other Sorting Algorithms.

The sorting problem admits a variety of solutions. We have detailed the synthesis of four - a selection sort, insertion sort, mergesort, and quicksort. In this section we briefly indicate how some of the other kinds of sorting algorithms could be synthezised by our method. The exercise of deriving several sorting algorithms from a single specification has also been reported in [6,7,10].

Bubble Sort and Sinking Sort [13]

Bubble sort can be viewed as a variation on selection sort in which the Select operation is performed by scanning through the input list interchanging consecutive elements which are out of order. The result is that the smallest element is deposited at one end of the list. The smallest element and the rest of the list are then easily obtained. The scan and interchange process is an instance of the general programming technique known as local search. Local search produces an output object by a sequence of small local modifications applied to an input object. In the Bubble/Selection operator the local modifications are the interchanges of consecutive elements which are out of order. Thus to synthesize a bubblesort we would proceed as with Ssort except that we would construct a local search algorithm for Select rather than a simple divide and conquer algorithm.

Similarly, sinking sorts may be viewed as variations on insertion sorts in which the specification for Insert has been satisfied by a local search algorithm.

Heap Sort [13]

The essence of heapsort is the heap data structure which allows fast operators for insertion of arbitrary elements and selection of the smallest element on the heap. In a sense heapsort is like a selection sort in that it repeatedly obtains the smallest remaining element on the heap and adds it to the output. Heap sort is a classic instance of a greedy algorithm [1]. Thus, a design theory for simple greedy algorithms would enable the synthesis of heapsort.

5. Conclusion

In this paper we have presented a framework for a top-down program synthesis system. The future success of systems of this kind depend on the development of design methods covering many classes of useful algorithms. We have taken a first step in that direction with design methods for simple divide and conquer algorithms and simple conditional programs. We are currently implementing a system which includes these design methods.

The main distinguishing feature of our approach is the ability to decompose a problem into subproblems, each described by an automatically derived specification. This ability depends on knowledge of the structure of divide and conquer algorithms (formulated in Theorem 1) and an engine for deriving preconditions. The need to precisely express the structure of divide and conquer algorithms has led to the notions of decomposition algebras, homomorphisms from decomposition algebras to composition algebras, and finally, the expression of the divide and conquer principle as a computational homomorphism. We are currently using these tools in exploring the structure of other classes of algorithms. A precondition engine is being implemented and currently can handle most of the derivations in this paper. We feel that the precondition problem will eventually take on a wider significance than its role in our system. Many of the tasks faced by computer science and AI may be usefully formulated in terms of theorem proving in a first-order logic. See for examples [9]. The greater flexibility and power of preconditions should enable us to greatly extend the set of tasks which can be precisely expressed and handled by deductive means.

Another distinguishing feature of our approach is the correct handling of incomplete specifications. A user can even omit input conditions knowing that (at a cost!) the system can derive input conditions for its product. As a result specifications are somewhat easier to create. Another advantage of incomplete specifications is that it makes it easier for the system to create specifications for subproblems.

Acknowledgements

Many thanks to Carol for her help in typing this manuscript.

Appendix

We list below the data structure knowledge required by the examples of this paper. It is grouped according to data types and includes operator signatures, typical composition algebras, and known theorems. All variables are implicitly universally quantified.

1. Natural Numbers ($\mathbb{N}$)

Known Functions:

$+: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$<, \leq, =, \neq, \geq, >: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

$Id: \mathbb{N} \rightarrow \mathbb{N}$

Known Theorems - let i, j, k vary over $\mathbb{N}$

n1. $i>0 \iff i+j > j$

n2. $i+i>j \Rightarrow i > (j \text{ div } 2)$

2. Lists of Natural Numbers (LIST($\mathbb{N}$))

Known Functions:

$First: LIST(\mathbb{N}) \rightarrow \mathbb{N}^+$

$Rest: LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})^+$

$Cons: \mathbb{N} \times LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$

$[First,Rest]: LIST(\mathbb{N}) \rightarrow \mathbb{N}^+ \times LIST(\mathbb{N})^+$

$Listsplit: LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N}) \times LIST(\mathbb{N})$

$Append: LIST(\mathbb{N}) \times LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$

$Id: LIST(\mathbb{N}) \rightarrow LIST(\mathbb{N})$

$Length: LIST(\mathbb{N}) \rightarrow \mathbb{N}$

$Ordered: LIST(\mathbb{N}) \rightarrow \mathbb{B}$

$Bag: LIST(\mathbb{N}) \rightarrow BAGS(\mathbb{N})$

Algebras: $\langle\{LIST(\mathbb{N}), \mathbb{N}\}, \{Cons\}\rangle$

$\langle\{LIST(\mathbb{N})\}, \{Append\}\rangle$

Known Theorems: let x, y, z vary over LIST(IN), i vary over IN, and w vary over BAGS(IN)

L1. Defined:x

L2. x=x

L3. Defined:x $\wedge$ Defined:y $\Leftrightarrow$ Defined:(x = y)

L4. $x \neq nil \Leftrightarrow$ Defined·First:x

L5. $x \neq nil \Leftrightarrow$ Defined·Rest:x

L6. $x \neq nil \Leftrightarrow$ Defined:[First,Rest]:x

L7. [First,Rest]:x = <a,y> $\Leftrightarrow$ Cons:<a,y> = x

L8. Bag·Cons:<a,x> = Add:<a,Bag:x>

L9. Bag·Append:$<x_1,x_2>$ = Union:$<$Bag:x_1,Bag:$x_2>$

L10. Defined·Listsplit:x

L11. Ordered:nil

L12. a $\leq$ Bag:x $\wedge$ Ordered:x $\Leftrightarrow$ Ordered·Cons:<a,x>

L13. Ordered:x_1 $\wedge$ Ordered:x_2 $\wedge$ $x_1 \leq x_2 \Leftrightarrow$ Ordered·Append:$<x_1,x_2>$

L14. Length:nil = 0

L15. 1 + Length:x = Length·Cons:<a,x>

L16. $x \neq nil \Rightarrow$ 1 + Length·Rest:x = Length:x

L17. Length:x_1 + Length:x_2 = Length·Append:$<x_1,x_2>$

L18. Card·Bag:x = Length:x

L19. Bag:x = w $\Rightarrow$ Length:x = Card:w

L20. Bag:nil = $\emptyset$

3. Bags of Natural Numbers (BAGS(IN))

Known Functions:

Add: IN $\times$ BAGS(IN) $\rightarrow$ BAGS(IN)

Union:BAGS(IN) $\times$ BAGS(IN) $\rightarrow$ BAGS(IN)

$\leq$: IN $\times$ BAGS(IN) $\rightarrow$ IB

$\leq$:BAGS(IN) $\times$ BAGS(IN) $\rightarrow$ IB

Card:BAGS(IN) $\rightarrow$ IN

Algebras: <{BAGS(IN),IN},{Add}>

Known Theorems: let w vary over BAGS(ℕ) and i vary over ℕ

B1. $w = w$

B2. $i \leq \emptyset$

B3. $i_1 \leq i_2 \wedge i_1 \leq w \Leftrightarrow i_1 \leq \text{Add}:\langle i_2, w\rangle$

B4. $i \leq w_1 \wedge i \leq w_2 \Leftrightarrow i \leq \text{Union}:\langle w_1, w_2\rangle$

B5. $\text{Card}\cdot\text{Union}:\langle w_1, w_2\rangle = \text{Card}:w_1 + \text{Card}:w_2$

B6. $\text{Card}\cdot\text{Add}:\langle a, w\rangle = 1 + \text{Card}:w$

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1983), Data Structures and Algorithms. Addison-Wesley Pub. Co., Reading MA, 1983.

2. Backus, J. (1978), Can Programming be Liberated from the von Neumann Style? A Functional Style of Programming and its Algebra of Programs. CACM 21, 8(1978), pp 613-641.

3. Barstow, D.R. (1979), Knowledge-Based Program Construction, Elsevier North-Holland Inc., New York, 1979.

4. Bibel, W. (1980), Syntax-directed, Semantics-Supported Program Synthesis, Art. Intell. 14, 3(Oct. 1980), 243-262.

5. Bledsoe, W. (1977), Nonresolution theorem proving, Art. Intell. 9(1), 1977, pp 1-35.

6. Clark K.L., and Darlington, J. (1980), Algorithm Classification through Synthesis, The Computer Journal 23, 1(1980), pp. 61-65.

7. Darlington, J. (1978), A synthesis of several sort programs. Acta Informatica 11, 1(1978), 1-30.

8. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ, 1976.

9. Green, C.C. (1969), Application of Theorem Proving to Problem Solving, Proc. First Int'l. Joint Conf. Art. Intell., 1969, pp 219-239.

10. Green, C.C., and Barstow, D.R. (1977), On Program Synthesis Knowledge, Art. Intell. 10, 3(1978), 241-279.

11. Goguen, J.A., Thatcher, J.W., and Wagner, E.G. Initial Algebra Semantics and Continuous Algebras, JACM 1(24), 1977, 68-95.

12. Goguen, J.A., Thatcher, J.W., and Wagner, E.G. (1978), An initial algebra approach to the specification, correctness, and implementation of abstract data types. in Current Trends in Programming Methodology, Vol. 4. R.T. Yeh, Ed., Prentice-Hall Inc., Englewood Cliffs, NJ, 1978, 80-149.

13. Knuth, D.E. (1969), The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley Pub. Co., Reading, Mass., 1969.

14. Loveland, D.W. (1978), Automated Theorem Proving: A Logical Basis. North Holland Pub. Co., New York, 1978.

15. Nilsson, N. (1971), Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Pub. Co., New York, 1971.

16. Manna, Z., and Waldinger, R.J. (1979), Synthesis: Dreams ⇒ Programs. IEEE Trans. Software Eng. SE-5, 4(1979), 294-328.

17. Manna, Z., and Waldinger, R.J. (1980), A Deductive Approach to Program Synthesis, ACM TOPLAS 2, 1(1980), 90-121.

18. Parnas, D.L. (1972), On the Criteria to be Used in Decomposing Systems into Modules, CACM 15, 12(1972), 220-225.

19. Smith, D.R. (1981), A Design for an Automatic Programming System, Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver, BC, Canada, August 1981, pp 1024-1027.

20. Smith, D.R. (1982), Derived Preconditions and Their Use in Program Synthesis, Sixth Conference on Automated Deduction, Ed. D.W. Loveland, Lecture Notes in Computer Science 138, Springer-Verlag, New York, 1982, pp 172-193.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Douglas R. Smith Code 52Sc Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40